Desarrollo de un Sistema Operativo para Raspberry Pi 2

Alejandro Cancelo Correia
Tomás Golomb Durán
Raúl Sánchez Montaño

Grado en Ingeniería Informática, Facultad de Informática, Universidad Complutense de Madrid

Trabajo Fin de Grado

Junio de 2020

Director:
José Luis Risco Martín

Índice general

Palabras clave

Palabras clave en Español

- **Entrada/Salida**
- **Memoria**
- **Procesamiento**
- **QEMU**
- **Fundación Raspberry Pi**
- **Raspberry Pi 2**
- **Sistema Operativo**

Keywords in English

- **Input/Output**
- **Memory**
- **Processing**
- **QEMU**
- **Raspberry Pi Foundation**
- **Raspberry Pi 2**
- **Operating System**

Resumen

Desarrollo de un Sistema Operativo para Raspberry Pi 2

Este trabajo extiende el deseo de la fundación Raspberry Pi de estimular la enseñanza de la informática, desarrollando un sistema operativo compacto para la Raspberry Pi 2 con el objetivo principal de reforzar la docencia en la Facultad de Informática de la Universidad Complutense de Madrid.

Se han desarrollado los módulos esenciales de un sistema operativo convencional que se pueden agrupar en tres bloques: memoria, procesamiento y entrada/salida. De esta forma, se busca facilitar y acelerar el trabajo de los docentes para la renovación de prácticas y contenido de las asignaturas troncales que son los pilares de todas las ramas de la ingeniería informática.

Asimismo, se abren las puertas a la creación de un ambiente de propósito educativo alrededor del Sistema Operativo dentro de la comunidad universitaria promulgándolo a través de otros trabajos de fin de grado que mejoren y amplíen éste. Además, se puede orientar a aplicaciones universitarias como un tablón de anuncios digital, dispositivos autónomos o sensores que miden la calidad del aire, temperatura o aforo.

Para ello se comparte el código fuente de *artOS* bajo la licencia MIT en el siguiente repositorio: https://github.com/dacya/tfg1920-raspiOS.

Development of an Operating System for the Raspberry Pi 2

This project extends the Raspberry Pi foundation's desire of encouraging the computer science education by developing an operating system for the Raspberry Pi 2 with the main objective of strengthen the education of the Computer Engineering Faculty.

It has been developed essential modules from a conventional Operating System, which can be grouped into three categories: memory, processing and input/output. This way, we are looking into making easier and speed up the task of the renovation of projects and fundamental subjects'syllabus, which are the fundamental pillars of all computer science degrees.

Furthermore, this project takes the first steps towards the creation of an educational environment and passing the responsibility of improving and expanding out to others bachelor's degree projects. In addition, the future development can be oriented to real applications such as a digital bulletin board, autonomous devices or humidity, air quality, temperature or capacity sensors.

To fulfill this goals, the code of *artOS* is shared under the MIT license in the following repository: https://github.com/dacya/tfg1920-raspiOS.

Capítulo 1

Introducción

Este trabajo de fin de grado se centra en el diseño e implementación de un sistema operativo para la Raspberry Pi 2, la segunda versión de la famosa serie de placas fabricada por la fundación inglesa Raspberry Pi [7].

El trabajo ha consistido en el desarrollo de los módulos esenciales que conforman un sistema operativo. Éstos se pueden agrupar en tres bloques:

- La **gestión de memoria** física es el bloque esencial que da soporte a la mayoría de los módulos. Los componentes se encargan de distribuir la memoria disponible para que puedan convivir el resto sin confrontamientos. Por ejemplo, los módulos de esta agrupación acomodan las necesidades de los procesos para dar lugar a la concurrencia sin colisiones de datos.
- El **control de procesamiento** se encarga de racionar la potencia del procesador a través de un sistema de procesos combinado con el sistema de interrupciones, pudiendo elegir entre distintos esquemas para adaptarse a las necesidades.
- La **entrada y salida** ofrece varias alternativas para la recepción y emisión de datos, por ejemplo: el banco de pines de la GPIO o la interfaz gráfica para el usuario disponible a través del puerto HDMI rompiendo así la dependencia con otro Sistema Operativo auxiliar.

1.1. Objetivos

El objetivo principal del proyecto es diseñar un sistema operativo de código abierto y con propósitos educativos para la Raspberry Pi 2, denominado artOS.

Todo el proyecto se pone a disposición de la comunidad para que se pueda usar como herramienta base en las asignaturas basadas en Sistemas Operativos con el fin de facilitar y agilizar el proceso de actualización y mejora del contenido y prácticas de éstas. El proyecto está estructurado para su fácil modificación, primando la sencillez de diseño y comprensión. Además, es importante que se pueda lanzar en un emulador, de tal forma que se puedan realizar fácilmente modificaciones, pruebas y depuraciones, por eso hemos elegido el emulador QEMU.

El sistema operativo desarrollado cumple con los siguientes requisitos:

- Soporte básico para la arquitectura ARM.
- Sistema mínimo de E/S, con capa de gestión de interrupciones y una interfaz básica de usuario.
- Ejecución multiproceso con cerrojos para permitir concurrencia.
- Gestión de memoria y sistema de ficheros con los métodos básicos.

1.2. Estado del arte

La fundación Raspberry Pi tiene el objetivo de estimular la enseñanza de la informática. Sin duda se ha conseguido considerando la existencia de este proyecto, pero, además, el dispositivo ha transcendido más allá resultando en un microprocesador de carácter general destacando por el uso extendido en la robótica gracias al banco de puertos que tienen todas las versiones a un lateral de la placa. Además, se ha desarrollado una gran comunidad alrededor de foros, blogs y páginas web para desarrollar múltiples herramientas de código abierto.

La primera Raspberry Pi se lanzó en febrero de 2012. Tras su lanzamiento, en junio de ese mismo año, se lanzó el primer SO completamente compatible con la infraestructura de la Raspberry Pi, cuyo nombre es Raspbian [15]. Además, el interés en SO desarrollados para Raspberry Pi ha ido aumentando gracias al auge de el IoT (del inglés Internet of Things), de manera que incluso Microsoft lanzó su propio SO [10] compatible con Raspberry Pi.

En cuanto al entorno educativo, existen algunos intentos de desarrollo libre con un propósito más académico, como el proyecto en el que nos hemos embarcado. Uno de ellos es de la Universidad de Cambridge [11], el cual está pensado para realizar el SO solamente en código ensamblador y para Raspberry Pi. El nuestro, sin embargo, está programado en C y ensamblador, por lo que es más legible. Además, hemos avanzado más allá en todos los sentidos ya que solo se presentan aplicaciones concretas y no un Sistema Operativo de carácter general.

Otro de los intentos, el cual nos ha servido de guía debido a que está pensado para la Raspberry Pi 2, es el desarrollado por Jake Sandler [17], trabajador de Google. Explica de manera muy fluida los procedimientos, aunque tiene varios fallos de diseño en el sistema de procesos, errores en el módulo de GPU (no funciona) o el uso de una estructura de una lista genérica en varios módulos con fallos de implementación. Además, hemos ido más allá que la visión de este intento desarrollando una interfaz de usuario, un sistema de ficheros y hemos mejorado la abstracción de los módulos, por ejemplo, junta el mailbox y el frambuffer en uno único.

Por último, también indicar que los dos intentos anteriores no se esfuerzan en mejorar la reusabilidad del código, mientras que nosotros hemos creado librerías e interfaces para facilitar el desarrollo futuro.

1.3. Metodología y plan de trabajo

Junto al director del trabajo, José L. Risco Martín, se decidió realizar reuniones periódicas en las que se marcaba el progreso y se planteaban las siguientes tareas a realizar. A raíz de la

pandemia del COVID-19 y el decreto de estado de alarma, se mantuvo el ritmo de reuniones pero telemáticamente a través de la aplicación web *Meet* de Google. Respecto a la comunicación entre miembros, se han utilizado aplicaciones de mensajería y antes de la pandemia se realizaron reuniones en las salas de la biblioteca y laboratorios de la Facultad de Informática.

Como el grupo está formado por tres componentes, se ha trabajado en paralelo centrándose cada alumno en una rama de desarrollo. Para ello, se ha hecho uso de la herramienta *git* para el control de versiones y de un repositorio en *GitHub* para guardar y gestionar nuestro progreso. Además, para facilitar el uso de los módulos se han documentado las funciones del código fuente expuestas utilizando la misma sintaxis definida por *JavaDoc*.

Respecto a esta memoria, cada miembro del equipo ha plasmado y explicado técnicamente su trabajo en el capítulo 2 de diseño. El resto de capítulos se han distribuido equitativamente para su desarrollo con varias iteraciones entre los alumnos y el director José L. Risco Martín.

1.4. Estructura de la memoria

Este documento recoge el proceso de desarrollo del sistema operativo, como compilarlo y ejecutarlo y las conclusiones finales. La distribución de capítulos es la siguiente:

- El capítulo 2 explica técnicamente el diseño con detalles de implementación y ejemplos de uso de los módulos que componen el sistema operativo.
- El capítulo 3 muestra como compilar, ejecutar el sistema operativo y las funcionalidades disponibles dentro de él.
- En el capítulo 4 se exponen las conclusiones finales y se indican posibles líneas de trabajo futuro.
- Finalmente, en el capítulo 5 se resumen las aportaciones realizadas por cada uno de los miembros del equipo.

Capítulo 2

Diseño

En este capítulo se muestra en detalle el diseño y la implementación de los distintos módulos que componen un sistema operativo moderno [18] además de ejemplos de uso. Para detallar los módulos, hemos utilizado el orden cronológico de implementación, ya que los últimos módulos implementados utilizan en gran medida módulos anteriores.

2.1. Introducción

Al desarrollar un sistema operativo, aparecen varias dependencias con el hardware que provocan la incompatibilidad entre dispositivos. En este proyecto se trabaja conociendo la arquitectura ARMv7 y, por ejemplo, el proceso de *arranque*, interrupciones y parte del sistema de procesos se implementan a bajo nivel con el repertorio de instrucciones propio del dispositivo [9]. Luego un cambio en el conjunto de instrucciones provoca la incompatibilidad de esos módulos como se ha comprobado en Raspberry Pi 3, la nueva versión, que tiene un *set* de instrucciones de 64 bits.

Por este motivo, y como este trabajo final de grado esta enfocado a la creación de un sistema operativo para la plataforma de Raspberry Pi 2 Model B, antes de empezar con el diseño conviene conocer las características mas destacables y con las que se va a interactuar de la placa:

- Un procesador basado en la arquitectura ARMv7, en concreto, el modelo Cortex-A7 a 900MHz de 32 bits.
- 1 GB de SDRAM (del inglés Synchronous Dynamic Random Access Memory).
- Puerto HDMI (del inglés High-Definition Multimedia Interface).
- Ranura para Micro SD (del inglés Secure Digital).
- Tarjeta gráfica VideoCore IV 3D graphics core.
- Banco de puertos GPIO (del inglés General Purpose Input/Output).

2.1.1. El Toolchain

En este capítulo se explicará el conjunto de herramientas necesarias para construir, depurar y ejecutar el proyecto de forma sencilla y automatizada.

En este proyecto se han usado y preparado para futuros desarrolladores las siguientes partes:

- Un Makefile con el que poder compilar y ejecutar el proyecto.
- Un compilador y un depurador, además de herramientas varias para poder analizar los ficheros resultados.
- Un emulador con el que poder probar el proyecto.

2.1.2. Estructura del *Makefile*

Los archivos *Makefile* son parseables gracias a la herramienta make del proyecto GNU [8], donde se puede encontrar una definición más precisa acerca de como funcionan los *Makefile*, y de cómo obtener la herramienta *make* en el caso de que no estuviera ya instalada en el sistema de trabajo.

Makefile del proyecto

En esta sección se procederá a mostrar el Makefile que se ha usado a lo largo de todo el desarrollo y a explicar en qué consiste cada regla y las variables declaradas.

De la linea 1 a la 10 del código 2.1 se declaran una serie de variables que sirven como argumentos para el compilador y el enlazador (línea 1 a la 5), para saber dónde se sitúa el código fuente (línea 8), dónde está el código en ensamblador (línea 9) y en qué carpeta guardar los archivos que generará el compilador al acabar su ejecución (línea 10).

```
1  CC = compiler/bin/arm-none-eabi
2  OBJCOPY = compiler/bin/arm-none-eabi-objcopy
3  C_OPT = -mcpu=cortex-a7 -O0 -Wall -Wextra -fpic -ffreestanding -std=
       gnu99 -nostdlib -I berryOS/include -g
4  #-g preserves the program's identifiers and symbols
5  L_OPT = -ffreestanding -nostdlib -mcpu=cortex-a7
6
7  #-fpic position independent code
8  SRC_DIR = berryOS/src
9  SRC_ARCH = berryOS/arch
10 BUILD_DIR = build
```

Código 2.1: Inicio de nuestro Makefile

Las opciones de compilación para el código c (variable C_OPT) son las siguientes:

- **-mcpu** sirve para indicar al compilador para qué procesador se quiere que se ejecute el código C.
- **-O0** le indica al compilador que no optimice código de ninguna forma. Esta opción acompañada junto con la opción -g facilita la depuración.

- **-Wall** y **-Wextra** sirve para que el compilador avise de cualquier tipo de posibles errores que encuentre durante las primeras fases de la compilación.
- **-fpic** permite al compilador generar código que sea independiente de donde esté localizado en memoria. Esto se traduce en que, por ejemplo, los cálculos de las direcciones de memoria a las que saltar para una instrucción de la forma *cmp* sean relativos en vez de absolutos.
- **-ffreestanding** obliga al compilador a trabajar en un entorno en el que es posible que la librería estándar no exista, y que el programa pueda no empezar en la función típica "main". Un entorno como este se traduce en que es posible que las funciones estándar pueden no tener su funcionalidad típica.
- **-std** simplemente fuerza a que el compilador siga los estándares definidos por *gnu99*.
- **-nostdlib** hace que el compilador no enlace las librerías estándar, acción que hace por defecto.
- **-I** simplemente indica dónde están los archivos .h que se quieren incluir.
- **-g** indica que se mantengan en el propio ejecutable ciertos símbolos fuente para poder depurar con *GDB*.

Las siguiente variables declaradas son arrays que contienen las rutas para cada archivo con extensión .c o .S que se quieren compilar. En caso de querer comprobar el contenido de estas variables, se puede ejecutar la regla **variable_test**.

Antes de continuar, es necesario mencionar que para poder obtener el valor de una variable, se debe escribir como *$(<nombre de variable>)*.

```
1  C_FILES = $(wildcard $(SRC_DIR)/*/*/*.c)
2  C_FILES += $(wildcard $(SRC_DIR)/*/*.c)
3  C_FILES += $(wildcard $(SRC_DIR)/*.c)
4
5  ASM_FILES = $(wildcard $(SRC_ARCH)/ARMv7/*.S) #Remember to add the
       context.S when using processes
6  ASM_FILES += $(wildcard $(SRC_DIR)/proc/*/*.S)
7  ASM_FILES += $(wildcard $(SRC_DIR)/proc/*.S)
8
9  OBJ_FILES = $(C_FILES:$(SRC_DIR)/%.c=$(BUILD_DIR)/%_c.o)
10
11 OBJ_FILES += $(ASM_FILES:$(SRC_DIR)/%.S=$(BUILD_DIR)/%_s.o)
```

A continuación se explican las reglas que son los puntos de partida de la herramienta *make*, de tal forma que pueden ser ejecutadas escribiendo en la terminal el comando *"make <nombre de la regla>"*.

Para que una regla se ejecute deben cumplirse todas sus dependencias. Si se quiere ejecutar la regla *build* del fragmento 2.2 primero debe cumplirse que el archivo *linker.ld*, cuya posición en

el repertorio de carpetas viene definido por el *path* que contiene una de las variables previamente mostradas, y todos los archivos con extensión .o, los archivos objeto creados por el compilador y necesarios para el enlazador.

```
build: $(SRC_ARCH)/ARMv7/linker.ld $(OBJ_FILES)
    @echo "Linking .o files..."
    $(CC)-gcc -T $(SRC_ARCH)/ARMv7/linker.ld -o $(BUILD_DIR)/myos.elf $
        (L_OPT) $(OBJ_FILES)
    #$(CC)-objcopy $(BUILD_DIR)/myos.elf -O binary $(BUILD_DIR)/myos.
       bin
    @echo "Done!"
```

Código 2.2: Regla build del Makefile

Nótese que las siguientes reglas garantizan que lo archivos .o existen una vez acabada la ejecución, por ejemplo:

```
#target for .c files
$(BUILD_DIR)/%_c.o: $(SRC_DIR)/%.c
    @mkdir -p $(@D)
    @echo "Compiling .c files..."
    $(CC)-gcc $(C_OPT) -MMD -c $< -o $@
    @echo ""

#target for .s files
$(BUILD_DIR)/%_s.o: $(SRC_DIR)/%.S
    @mkdir -p $(@D)
    @echo "Compiling .S files..."
    $(CC)-gcc $(C_OPT) -MMD -c $< -o $@
    @echo ""$(SRC_DIR)/
```

El resto de reglas sirven para:

- La regla **run** para iniciar todo el proceso de compilar, enlazar y ejecutar el programa final.
- Realizar el mismo proceso que la regla anterior, salvo que esta vez el programa se ejecutará y esperará a que haya un GDB para continuar el proceso. De esta forma, podemos depurar el código con la regla **debug**.
- Las reglas **build** y **build_hard** permiten únicamente compilar y enlazar el programa. En el caso de la última, genera la imagen del sistema operativo listo para ser cargado en la tarjeta SD de la Raspberry Pi 2.
- La regla **clean** borra todo los archivos que se crean con las ejecuciones de las reglas **build** o **build_hard**.

```
run: build
    @echo ""
    @echo "Running qemu..."
```

```
4       qemu-system-arm -m 256 -M raspi2 -serial stdio -kernel $(BUILD_DIR)
            /myos.elf
5
6   debug: build
7       @echo ""
8       @echo "Debugging in qemu..."
9       qemu-system-arm -m 256 -M raspi2 -serial stdio -kernel $(BUILD_DIR)
            /myos.elf -S -gdb tcp::1234
10
11  build: $(SRC_ARCH)/ARMv7/linker.ld $(OBJ_FILES)
12      @echo "Linking .o files..."
13      $(CC)-gcc -T $(SRC_ARCH)/ARMv7/linker.ld -o $(BUILD_DIR)/myos.elf $
            (L_OPT) $(OBJ_FILES)
14      #$(CC)-objcopy $(BUILD_DIR)/myos.elf -O binary $(BUILD_DIR)/myos.
            bin
15      @echo "Done!"
16
17  build_hard: build
18      $(OBJCOPY) $(BUILD_DIR)/myos.elf -O binary $(BUILD_DIR)/kernel7.img
19      @echo "Done!"
20  clean:
21      rm -rf $(BUILD_DIR)/
```

Código 2.3: Regla de ejemplo

2.1.3. El compilador

El compilador empleado pertenece a la *toolchain* de *arm-none-eabi*. Este conjunto de herramientas se enfoca en la generación de código para procesadores *ARM*, es open source, está enfocado en lo que se conoce como sistemas *bare metal* y compila con *ARM-EABI*. Se puede encontrar en la pagina oficial de ARM [4], aunque también se incluye este paquete en el repositorio del proyecto en la carpeta denominada **compiler/**.

Dentro de la carpeta **compiler/bin/** se encuentran todos los programas que se han usado para el desarrollo. Los más frecuentes son el programa *GDB* con el que depurar el código y el compilador *GCC* cuestión.

2.1.4. El emulador

Se decidió emplear el emulador de **QEMU** debido a que no todos los miembros del equipo tienen en posesión una Raspberry Pi 2 con la que trabajar, y a la rapidez que ofrece para probar el código.

Además, el emulador ha servido para poder continuar con el trabajo durante el confinamiento obligatorio decretado en el estado de alarma a causa de la pandemia de COVID-19. Es por el mismo motivo, el cual no se ha podido depurar el código al completo en la placa pero sí en el emulador.

Para instalar el emulador QEMU en Linux se consigue con el siguiente comando:

```
1   sudo apt-get install qemu
```

2.2. Arranque del sistema

La Raspberry Pi 2 tiene una forma muy peculiar de realizar la etapa de *bootloading*, por lo que en esta sección hablaremos de forma breve acerca de ella. Antes de comenzar, debemos mencionar que la Raspberry Pi tiene el kernel del sistema operativo, junto con otros programas básicos, en la tarjeta micro SD.

Para obtener todos los archivos y el firmware necesarios en la SD, seguiremos el siguiente proceso:

1. Instalaremos Raspbian en la SD siguiendo este tutorial oficial [6]. Nosostros hemos utilizado Raspberry Pi Imager. Tras la instalación, tendremos la tarjeta dividida en dos carpetas, *boot* y *rootfs*.

2. Insertamos la SD en la Raspberry Pi 2 y conectamos el HDMI para comprobar que se inicia Raspbian, lo que indica que la instalación se ha realizado correctamente.

3. Compilaremos el proyecto utilizando *make build_hard* (más información sobre cómo utilizar los comandos del Makefile en la sección 3.1). De esta manera, obtendremos en la carpeta *build* el archivo *kernel7.img*.

4. Finalmente, borramos de la carpeta *boot* todos los archivos *.img, y copiamos el archivo del paso anterior en su lugar.

El bootloading se podría dividir en cinco etapas. No entraremos en demasiado detalle acerca de estas etapas, ya que lo que se hace dentro de cada una de ellas no es conocimiento 100 % abierto al público.

Las cinco etapas son:

1. Se ejecuta el firmware de la ROM, este software cargará la siguiente etapa en la cache L2. El componente que ejecuta el programa es la GPU, la CPU permanece deshabilitada durante todo el proceso hasta la última fase. La SDRAM también está desactivada.

2. Esta etapa consiste en ejecutar el código dentro del archivo *bootcode.bin* que está dentro de la micro SD y habilitar la SDRAM para poder cargar la siguiente etapa.

3. Lo que se ejecutará en esta etapa es el código del archivo *loader.bin*, este archivo es capaz de parsear los ficheros con formato .elf y será el que cargue el fichero *start.elf* para la siguiente etapa.

4. *start.elf* cargará el fichero llamado *kernel.img* y leerá los ficheros *config.txt*, *cmdline.text* y *bcm2835.dtb*.
   Si el fichero con extensión .dtb existe, se cargará en la posición de memoria 0x100 y el kernel en la posición 0x8000.

Si la línea *disable_commandline_tags=true* está dentro del fichero *config.txt*, el kernel se carga en la posición 0x0.
En cualquier otro caso, el kernel siempre se cargará en la posición 0x8000 y los ATAGS se añadirán a partir de la 0x100.

5. A partir de esta etapa es cuando el sistema operativo gana control del hardware y se empieza a ejecutar su kernel.

Para ayudar a entender este proceso más fácilmente, observa la figura 2.1.

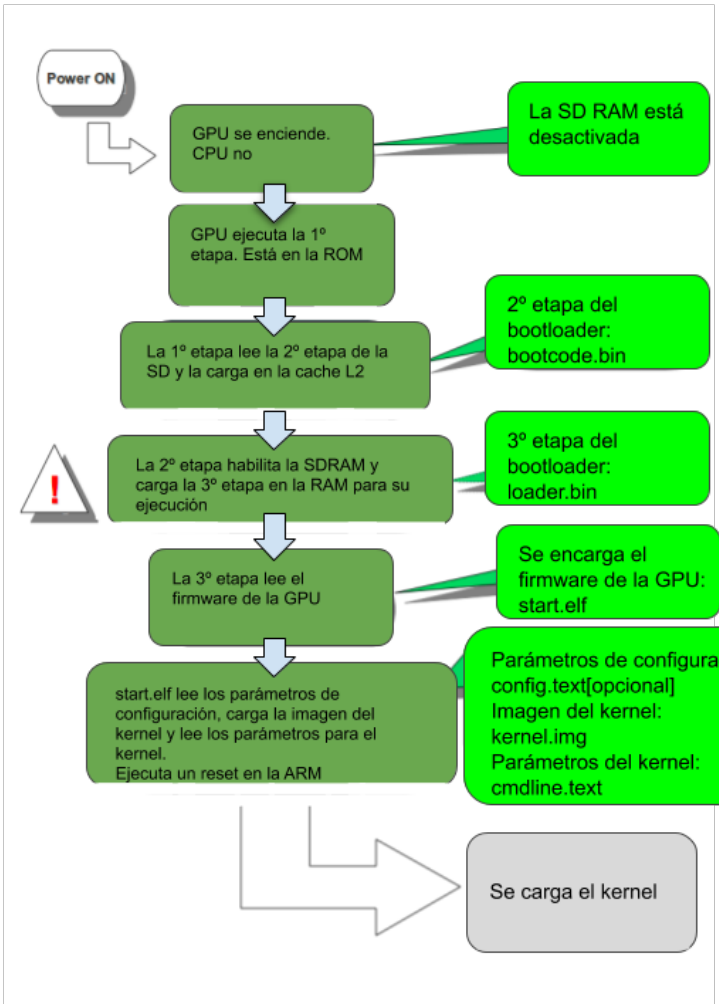


Figura 2.1: Diagrama del bootloading

2.3. Kernel

El kernel [5] es lo primero que se ejecutará tras el proceso de arranque y se encargará de realizar las acciones necesarias para ajustar ciertas partes e inicializar los distintos módulos. De esto se encargarán *berryOS/arch/ARMv7/boot.S* y *berryOS/src/kernel.c* respectivamente. Los procesos de *boot.S* son los siguientes:

- Pausar 3 de los 4 procesadores para conseguir un sistema monoprocesador.
- Colocar ceros en toda la sección de BSS(del inglés Block Started by Symbol). Esto es necesario para seguir el estándar de C, en el que las variables globales sin inicializar deben valer cero.
- Cambiar a los distintos modos de ejecución para inicializar sus registros. Básicamente utilizaremos dos: el modo IRQ (del inglés Interrupt Request) y el supervisor.
- Inicializar la tabla de vectores de excepciones con las funciones que las tratarán. De esta manera, cuando salte una excepción, se saltará directamente a la función que la tratará. La función *str_vect_table* que se encuentra en *berryOS/arch/ARMv7/interrupt_init.S*, es la encargada de esto, situando la tabla a partir de la posición de memoria cero.
- Habilitar las interrupciones.
- Llamar a la función *kernel_main* de kernel.c

En esta misma sección se pueden realizar otras acciones, las cuales no hemos utilizado porque no eran básicas, pero se podrían considerar en el futuro:

- La MMU(del inglés Memory Management Unit) para la gestión de memoria virtual.
- La inicialización de dispositivos de entrada/salida.
- La inicialización de NEON, relacionado con la implementación del SIMD (del inglés Single Instruction Multiple Data) avanzado, y VFP (del inglés Vector Float Point), la extensión para vectores de punto-flotante.
- La utilización de Secure World para iniciar la Raspberry Pi en modo seguro.

El método *kernel_main*, inicializará cada uno de los módulos que lo necesiten, llamando a su función init. Estos son:

- **Modulo de entrada/salida**: compuesto por la GPIO, la UART (del inglés Universal Asynchronous Receiver-Rransmitter) y la GPU (del inglés Graphics Process Unit).
- **Modulo de memoria**: compuesto por el proceso de mapeado de la memoria y el gestor de memoria dinámica.
- **Modulo de procesos**: incluye los procedimientos para realizar cambios entre procesos y el mecanismo para registrar planificadores.
- **Comandos**: inicializa los mecanismos necesarios para crear y registrar comandos.
- **Consola**: la cual contiene la interfaz con la que se comunica el usuario.
- **Interrupciones**: compuesto por las funciones que tratan las distintas interrupciones del sistema.
- **Timer local**: encargado de inicializar el timer local, de manera que salte cada un tiempo determinado.

2.4. Entrada / Salida

Esta sección explica los módulos de entrada y salida en el orden temporal en que se han ido desarrollando, así como a los módulos dependientes disponibles en el momento y las necesidades que se presentan, por ejemplo, si se está trabajando en la Raspberry o en el emulador QEMU. Esto se traduce en un incremento de complejidad con cada nuevo módulo, pero que responde con una mejor experiencia de usuario y, en el proceso de desarrollo del proyecto, proporciona una depuración y testeo más rápido y eficaz. Este esfuerzo culmina en una librería estándar de entrada/salida similar a la conocida popularmente como *stdio.h*.

2.4.1. GPIO

Este módulo tiene como objetivo simplificar el uso de una de las características más famosas de la serie de placas Raspberry: los pines de Entrada/Salida de Propósito General (*General Purpose Input/Output pins (GPIO)* en inglés).

El banco de pines GPIO se utiliza como puerto para la conexión de periféricos hardware con la CPU. Cada pin tiene varias funcionalidades disponibles y son manipulables por medio de una lista de registros bien definidos por el fabricante[1] [12]. En la figura 2.2 se muestra la disposición de los pines.

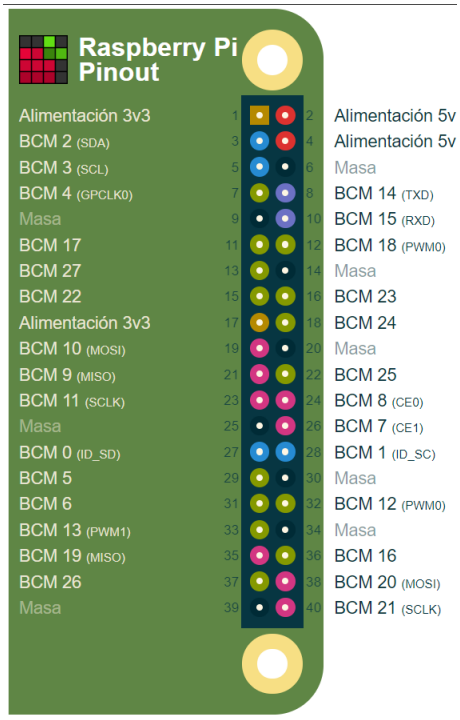


Figura 2.2: Imagen que ofrece https://es.pinout.xyz/ con la información de cada pin

[1] La lista completa se puede consultar en la sección **6.2** del manual de periféricos https://github.com/raspberrypi/documentation/files/1888662/BCM2837-ARM-Peripherals.-.Revised.-.V2-1.pdf

Mapeo de pines

```
#define GPIO_BASE    (PHYSICAL_PBASE + 0x00200000)
```

Código 2.4: Constante para mapeo de los pines GPIO

La arquitectura de la Raspberry Pi 2 mapea la memoria dejando en el rango de 0x3F000000-0x3FFFFFFF las direcciones que usarán los periféricos de Entrada/Salida, en concreto los pines GPIO. Estos valores varían si se usa memoria virtual, y dado que en este proyecto no ha sido implementada, por eso se facilita dicho rango. Como se muestra en el fragmento de código 2.4, la constante global *PHYSICAL_PBASE* tiene la dirección inicial 0x3F000000 y en este módulo se utiliza para calcular las direcciones de los registros de manipulación de los pines GPIO como se especifica en la documentación.

```
//GPIO function select (registers 0 to 9)
#define GPFSEL0     ((volatile uint32_t*)(GPIO_BASE + 0x000))
//GPIO function select (registers 10 to 19)
#define GPFSEL1     ((volatile uint32_t*)(GPIO_BASE + 0x004))
//GPIO function select (registers 20 to 29)
#define GPFSEL2     ((volatile uint32_t*)(GPIO_BASE + 0x008))
//GPIO function select (registers 30 to 39)
#define GPFSEL3     ((volatile uint32_t*)(GPIO_BASE + 0x00C))
//GPIO function select (registers 40 to 49)
#define GPFSEL4     ((volatile uint32_t*)(GPIO_BASE + 0x010))
//GPIO function select (registers 50 to 53)
#define GPFSEL5     ((volatile uint32_t*)(GPIO_BASE + 0x014))
```

Código 2.5: Registros de manipulación GPIO de */include/io/gpio.h*

El fragmento de código anterior muestra parte de la lista de los registros de manipulación de los pines. Son las direcciones de memoria de los registros de 32 bits, que se marcan como volatile para esquivar la cache y evitar problemas de transmisión de datos.

Manipulación de los pines

La escritura y lectura de los registros ya mencionados es a nivel binario, por tanto es necesario conocer lo operadores binarios (llamados *bitwise operators* en inglés) nativos de C para entender la explicación y, consecuentemente, el código.

- **[0b-número-]** Indica que el formato de -número- es binario.
  Ejemplo: 0b1000 = 8
- **[&]** Operador AND lógico a nivel binario.
  Ejemplo: 0b1 & 0b0 = 0b0
- **[|]** Operador OR lógico a nivel binario.
  Ejemplo: 0b1 | 0b0 = 0b1

- **[< <]** Operador de desplazamiento[2] hacia la izquierda.
  Ejemplo: 0b1 < < 3 = 0b1000
- **[> >]** Operador de desplazamiento hacia la derecha.
  Ejemplo: 0b1000 > > 3 = 0b1

Escritura en un pin

```
typedef enum {
    INPUT  = 0b000,
    OUTPUT = 0b001,
    ALT0   = 0b100,
    ALT1   = 0b101,
    ALT2   = 0b110,
    ALT3   = 0b111,
    ALT4   = 0b011,
    ALT5   = 0b010
} pin_alt_funct;

void pin_set_function(unsigned int pin, pin_alt_funct fun_sel) {
    if (pin > 53) {
        return ;
    }

    volatile uint32_t* reg_obj;

    switch (pin / 10) {
        case 0:
            reg_obj = GPFSEL0;
            break;
        case 1:
            reg_obj = GPFSEL1;
            break;
        [...]
            reg_obj = GPFSEL4;
            break;
        case 5:
            reg_obj = GPFSEL5;
            break;
    }

    *reg_obj &= ~(0b111 << ((pin %10) * 3));
    *reg_obj |= fun_sel << ((pin %10) * 3);
}
```

Código 2.6: Configuración de los pines */src/io/gpio.c*

En el fragmento del código 2.6 se muestra cómo configurar la funcionalidad o forma de trabajar de un pin. Por ejemplo, un pin se puede utilizar para datos de entrada, para salida o para controlar eventos (por ejemplo si un pin toma cierto valor en algún momento del ciclo de reloj).

Los registros se dividen en *paquetes* de bits únicos para cada pin, en el ejemplo, cada registro se divide en paquetes de 3 bits que debe tomar alguno de los valores que se definen en el enumerado.

[2] Los operadores de desplazamiento también son útiles (y más eficientes) para calcular multiplicaciones por 2 (hacia la izquierda) y divisiones enteras entre 2 (hacia la derecha)

Con matemáticas no tan complejas, se puede calcular fácilmente el registro y la posición donde se debe escribir en este.

Por lo general, el esquema para escribir en un registro ARM es el siguiente:

1. **Calcular el registro** donde se encuentra el pin dividiendo entre el número de pines que entran por registro. [Líneas 19-32 del código 2.6]

2. **Limpiar** el valor que hubiese con anterioridad en el paquete de bits. Desplazando tantos 1s como el tamaño del paquete y posteriormente negarlo para dejar a 0 únicamente las posiciones de los bits del paquete al aplicar el AND.[3] [Línea 34 del código 2.6]

3. **Asignar** el valor aplicando al contenido del registro una OR con el nuevo dato desplazado hasta donde el paquete va. [Línea 35 del código 2.6]

En la figura 2.3 se muestra de forma visual el proceso recién comentado y el efecto que se producen en los paquetes de bits.

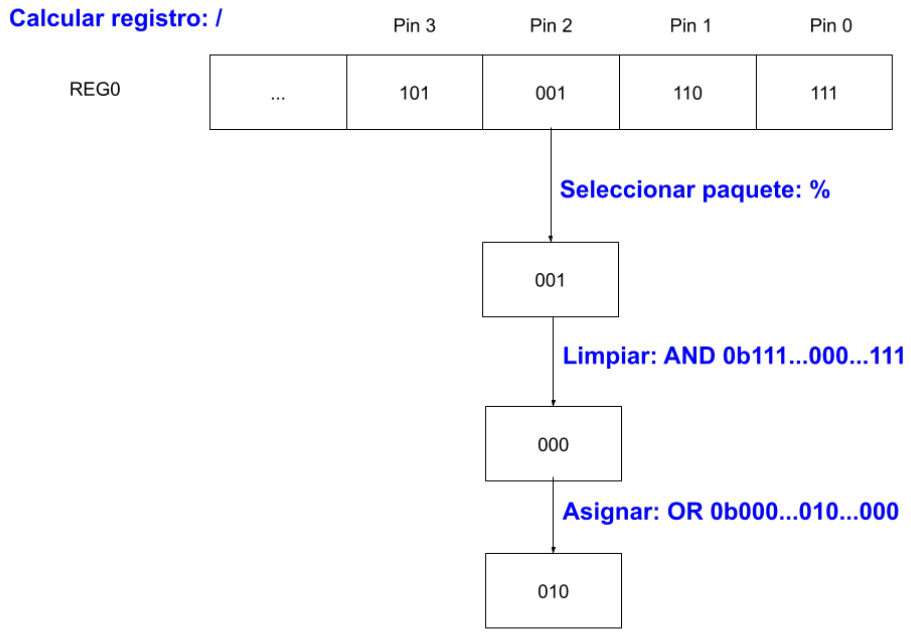


Figura 2.3: Esquema visual de escritura en un registro

Lectura de un pin

```
1  int pin_get_level(unsigned int pin) {
2      if (pin > 53)
3          return -1;
4
```

[3] En ocasiones este paso se puede omitir porque ARM limpia ciertos registros por ciclo de reloj o, dicho de otra forma, una vez usados

```
5       volatile uint32_t* reg;
6
7       switch (pin / 32) {
8           case 0:
9               reg = GPLEV0;
10              break;
11          case 1:
12              reg = GPLEV1;
13              break;
14      }
15
16      // Acorde al esquema explicado a continuacion
17      uint32_t valor = *reg;
18      valor = valor >> (pin % 32);
19      valor &= 0b1;
20
21      // Simplificado
22      return ((*reg) >> (pin % 32)) & 1;
23  }
```

Código 2.7: Lectura de un pin */src/io/gpio.c*

La rutina del código 2.7 muestra la implementación para leer el valor actual de un pin. Devuelve 1 para el estado *high*, que corresponde a un voltaje positivo a nivel hardware, y 0 para *low* y que se traduce en 0 voltios.

Igual que en la escritura, el siguiente esquema sirve para leer de un registro el valor actual del pin:

1. **Calcular el registro** donde se encuentra el paquete de bits correspondiente al pin. [Líneas 7-14 del ejemplo. 2.7]
2. **Guardar el valor** del registro usando el operador * de acceso a puntero. [Líneas 17 del ejemplo 2.7]
3. **Desplazar** el valor al principio, eliminando así los datos de los pines anteriores. [Línea 18 del ejemplo 2.7]
4. **Limpiar** el valor del resto de pines a la derecha que no interesan. [Línea 19 del ejemplo 2.7]

En la figura 2.3 se muestra de forma visual el proceso recién comentado y el efecto que se producen en los paquetes de bits para conseguir el resultado.

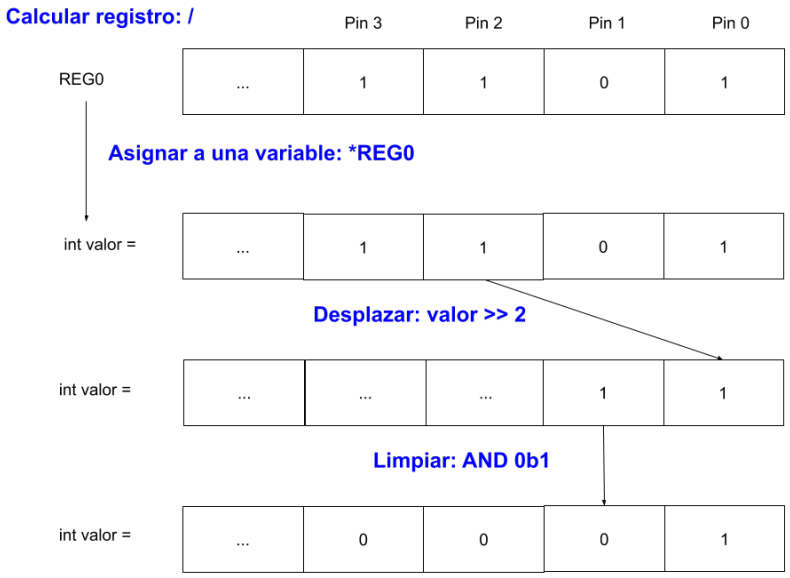


Figura 2.4: Esquema visual de escritura en un registro

Ejemplo de uso

A continuación se muestra de forma abstracta las principales operaciones desarrolladas en la librería sobre uno de los puerto del banco GPIO: Configurar el puerto como salida, liberar voltaje por el puerto, leer el estado de uno de los pines y parar la salida de voltaje.

```
1
2  #include <io/gpio.h>
3
4  // Se configura el pin 11 como salida
5  pin_set_function(11, OUTPUT);
6
7  // Se envia voltaje al puerto 11
8  pin_set_output(11);
9
10 // El pin devuelve un 1 y por tanto emite voltaje
11 int current_state = pin_get_level(11);
12
13 // Se corta el voltaje
14 pin_clear_output(11);
```

Código 2.8: Ejemplo de uso con io/gpio.c

Como *prueba de trabajo*, se adjunta la foto 2.5 que tomó el equipo mientras depuraba en la Raspberry Pi 2 cedida por la Universidad Complutense de Madrid para comprobar que el sistema operativo realizaba el proceso de *arranque* correctamente.

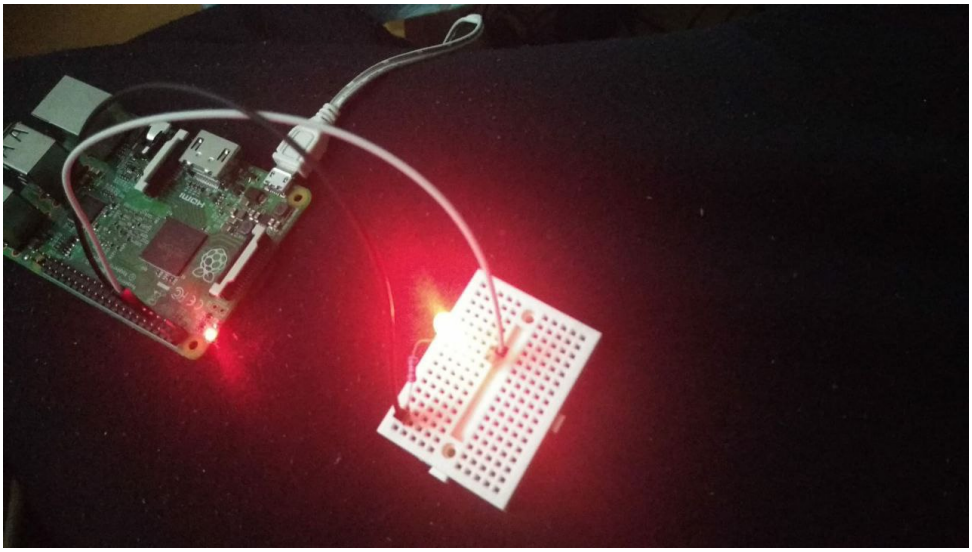


Figura 2.5: Depurando con un LED y la Raspberry Pi 2 cedida por la UCM.

2.4.2. UART

La UART es un dispositivo incorporado en la placa que sirve de protocolo para transmitir información de carácter general entre dispositivos. En el caso del emulador QEMU, se utiliza exclusivamente para imprimir por la terminal. En la Raspberry Pi 2 se utiliza a través de los pines GPIO 14 y 15 configurados correctamente y mediante un cable TTL como el de la figura 2.6. A nivel software, se resume en leer y escribir de un registro mapeado en memoria de forma similar a la explicada en la sección 2.4.1.

Figura 2.6: Cable TTL pin a USB. https://www.amazon.es/ADAFRUIT-Cable-serie-puntos-directa/dp/B00DJUHGHI

De la misma forma que en el módulo de la GPIO, se deben mapear los registros a partir de la dirección física y que cambia si se activa la memoria virtual. Se puede ver la definición de la dirección base en el siguiente código:

```
#define UART_BASE  (PHYSICAL_PBASE + 0x00201000)

#define UART0_DR ((volatile uint32_t *)(UART_BASE + 0x000))
[...]
```

Inicialización

En el caso de estar utilizando el emulador QEMU, nos dimos cuenta de que no es necesario realizar ningún proceso de inicialización, presumiblemente porque ya se realiza automáticamente al ser el único *puerto* de comunicación entre programa/emulador y el sistema donde se ejecuta. Sin embargo, para la Raspberry Pi 2 se debe proceder como se explicará a continuación.

0. Desactivar la UART.

```
*(UART0_CR) = 0;
```

1. **Configurar los pines** asignando a la alternativa de funcionalidad 5 que corresponde a la de la UART.

```
pin_set_function(14, ALT5);
pin_set_function(15, ALT5);
```

2. **Desactivar las resistencias pull-up/pull-down (pud)**. Estas resistencias sirven para evitar el efecto denominado *floating pin* que ocurre en pines de entrada sin conectar y que provoca ruido eléctrico, en este caso, se desactivan porque los pines de entrada estarán enchufados al cable TTL.

```
pin_switch_pud(14, 0b00);
pin_switch_pud(15, 0b00);
```

3. Por último se **asignan varios parámetros del protocolo**, como por ejemplo: el baudio, se activa la transmisión y la recepción, el tamaño de palabra (a 8 bits = 1 Byte = 1 char), tratamiento de errores y reactivar la UART.

```
//using baud as 115200, INTEGER = 3000000 / (16*115200) = 1.627 ~ 1
*(UART0_IBRD) = 1;

    [...]

selector = 0;
selector = (7<<4); //8 bits each word and FIFO enable
*(UART0_LCRH) = selector;

    [...]

selector = 0;
selector |= (1<<9); //receive enable
selector |= (1<<8); //transmit enable
selector |= 1; //uart enable

*(UART0_CR) = selector;
```

Lectura

La lectura de la UART es muy simple, únicamente se trata de leer de un registro, pero se debe saber cuando. Para eso, a través de un registro de control se comprueba que se haya detectado un dato en un while. Por tanto se trata de una *operación bloqueante activa síncrona*.

```
char uart_recv() {
    while ( *(UART0_FR) & (1 << 4) );

    return *(UART0_DR);
}
```

Código 2.9: Rutina de lectura de un char en la UART

Escritura

De la misma forma que la lectura, todo se gestiona a través de los registros pertinentes. Antes de leer se debe esperar a que la UART esté disponible, por tanto, igual que la lectura, se trata de una *operación bloqueante activa síncrona.*

```
void uart_putc(unsigned char c) {
    while ( *(UART0_FR) & (1 << 5) );

   *(UART0_DR) = c;
}
```

Código 2.10: Rutina de escritura de un char en la UART

Ejemplos de uso

Las dos sencillas operaciones ya explicadas sirven como pilares para implementaciones más complejas:

```
void uart_puts(const char* str) {
    // Llamar a la rutina para cada char del array
    for (size_t i = 0; str[i] != '\0'; i++)
        uart_putc((unsigned char)str[i]);
}

uart_puts("Hola lector!");
```

Código 2.11: Rutina de escritura de un string (char *) en la UART

```
void uart_hex_puts(uint32_t value) {
    char str_argument[9] = {'0','0','0','0','0','0','0','0', '\0'};
    convert_to_str(value, str_argument, 8);
    uart_puts("0x");
    uart_puts(str_argument);
    uart_puts("\r\n");
}

const char* hello = 'Hola lector!'
```

```
10
11 uart_hex_puts(hello);
12 /* Salida representativa: 0x00014CA0 */
```

Código 2.12: Rutina de escritura de una dirección de memoria en la UART

Figura 2.7: Salida por consola del arranque del SO emulado en QEMU a través de la UART

2.4.3. Mailbox, framebuffer y GPU

Esta sección explica cómo conseguir la interacción de la CPU con la GPU para poder *renderizar* imágenes por el puerto HDMI. Se explican estos tres módulos juntos porque son necesarios para lograr el objetivo. El proceso es complejo y se dificulta aún más al estar incompleta la documentación proporcionada por el fabricante[4].

El fin del proceso es conseguir un framebuffer, un array de píxeles compartido entre la GPU y la CPU en memoria donde es posible la lectura y escritura. Los datos que contiene se enviarán por el puerto HDMI en cada fotograma y por tanto renderizable en cualquier pantalla conectada.

Mailbox

El mailbox es un sistema que *facilita* el intercambio de datos por mensajes entre la CPU ARM y los dispositivos exteriores al chip, los *periféricos*, entre ellos está la tarjeta gráfica. Tiene varios canales, aunque en este proyecto únicamente se ha usado el octavo y por eso sólo se han modelado la estructuras de mensajes para éste, pero, definiendo las estructuras pertinentes, el módulo debería servir para cualquiera ya que se ha conseguido la abstracción necesaria para el envío y recepción.

Lista de canales mailbox:

- 0: Power management
- 1: Framebuffer

[4]La documentación oficial proporcionada por el fabricante de los distintos canales mailbox https://github.com/raspberrypi/firmware/wiki/Mailboxes

- 2: Virtual UART
- 3: VCHIQ
- 4: LEDs
- 5: Buttons
- 6: Touch screen
- 7: [5]
- 8: Property tags (ARM ->VC)
- 9: Property tags (VC ->ARM)

Como ya se ha explicado varias veces en este capítulo de entrada/salida, la lectura y escritura se realiza por medio de registros mapeados en memoria. En este caso, los datos en los registros ya no se tratan a nivel binario, sino que son estructuras con un formato definido como se muestra en la figura 2.8.

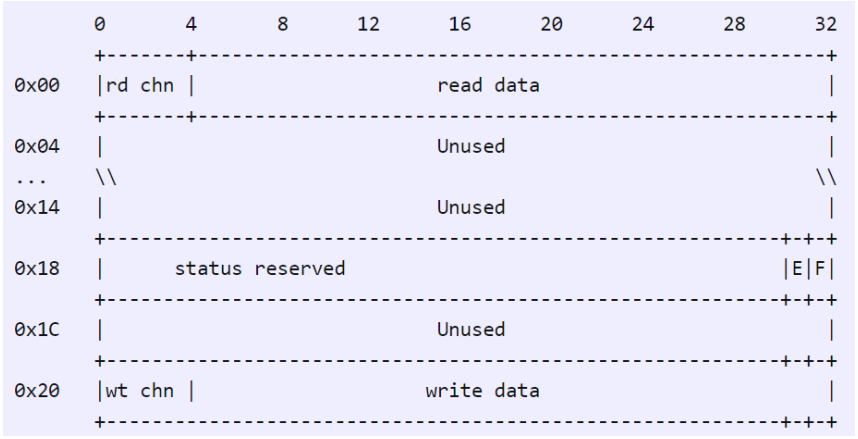


Figura 2.8: Estructura del mapeo de registros https://jsandler18.github.io/extra/mailbox.html

La figura 2.8 se refleja en código como el *struct* del fragmento 2.13. Posteriormente se definen los punteros a los registros de datos que van a tener esta forma.

```
typedef struct {
    uint8_t channel: 4; // Canal de la mailbox
    uint32_t data: 28;  // Direccion del mensaje
} mailbox_message_t;

#define MAILBOX_BASE PHYSICAL_PBASE + MAILBOX_OFFSET
#define MAIL0_READ (((mailbox_message_t *)(0x00 + MAILBOX_BASE)))
#define MAIL0_STATUS (((mailbox_status_t *)(0x18 + MAILBOX_BASE)))
#define MAIL0_WRITE (((mailbox_message_t *)(0x20 + MAILBOX_BASE)))
```

Código 2.13: Definición de registros de la mailbox

[5] Como ya se ha comentado, la documentación compartida por el fabricante en esta área es insuficiente

Recepción/lectura

```
/**
 * Returns the size for a mailbox message tag as
 * defined in the protocol.
 *
 * @param tag the tag to get the size of
 * @return bytes of a mailbox message type
 */
mailbox_message_t mailbox_read(mailbox_channel_t channel) {
    mailbox_status_t stat;
    mailbox_message_t res;

    // Make sure that the message is from the right channel
    do {
        // Make sure there is mail to recieve
        do {
            stat = *MAIL0_STATUS;
        } while (stat.empty);

        // Get the message
        res = *MAIL0_READ;
    } while (res.channel != channel);

    return res;
}
```

Código 2.14: Rutina de recepción de mensaje por un canal mailbox

Como se ve en el fragmento del código 2.14, la lectura es similar a las vistas con anterioridad. Es una operación *síncrona, bloqueante* y con *espera activa* para comprobar que la cola tiene mensajes así como la pertenencia al canal correspondiente. Devuelve el puntero a la estructura del mensaje recibido.

Emisión/escritura

```
void mailbox_send(mailbox_message_t msg, mailbox_channel_t channel) {
    mailbox_status_t stat;
    msg.channel = channel;

    // Make sure you can send mail
    do {
        stat = *MAIL0_STATUS;
    } while (stat.full);

    // send the message
    *MAIL0_WRITE = msg;
}
```

Código 2.15: Rutina de emisión de mensaje por un canal mailbox

De la misma forma que la recepción de datos del código 2.14, se trata de una *operación síncrona, bloqueante y de espera activa*. Simplemente se debe escribir el puntero del mensaje en el registro de escritura comprobando que la cola de mensajes no esté llena como se puede ver en el fragmento de código 2.15.

Obtención de framebuffer

Ahora que ya se conoce el protocolo de comunicación mailbox, el algoritmo a grandes rasgos[6] sigue los siguientes pasos:

1. Enviar un mensaje con varios parámetros de **configuración para el framebuffer** a la GPU a través del canal *MAILBOX_PROPERTY_CHANNEL*.

2. Enviar un mensaje de **instanciación de framebuffer** a la GPU a través del canal *MAILBOX_PROPERTY_CHANNEL*.

3. De no haberse producido ningún error, el último mensaje enviado tendrá el **puntero de memoria al framebuffer**, esto es posible porque las estructuras de los mensajes se pasan por referencia.

GPU

La inicialización de este módulo consiste en obtener el puntero del *framebuffer* explicado anteriormente y almacenarlo en una variable global para su uso. Cuando se haya conseguido esto, la librería está preparada para leer y escribir en cada *píxel* que se verá por la pantalla conectada al puerto HDMI de la Raspberry o en una ventana del emulador QEMU.

Un *píxel* es un número de 24 bits[7] que representa un color siguiendo el formato estándar RGB con 8 bits para cada color. En total admite $2^{24} = 16{,}777{,}216$ colores. La definición que se ha usado es la siguiente:

```
typedef struct {
    uint8_t r;
    uint8_t g;
    uint8_t b;
} color_24;

#define BLACK ((color_24){0x00, 0x00, 0x00})
#define WHITE ((color_24){0xFF, 0xFF, 0xFF})
#define RED   ((color_24){0xFF, 0x00, 0x00})
```

El proceso para colorear un un píxel es tan fácil como copiarlo en la posición correspondiente siempre que esté dentro de los límites del framebuffer. La rutina de escritura es la siguiente:

```
void write_pixel(uint32_t x, uint32_t y, color_24* pix) {
    if (x < fbinfo.width && y < fbinfo.height) {
        uint8_t* location = fbinfo.buf + y*fbinfo.pitch + x*BYTES_PER_PIXEL;
        memcpy(location, pix, BYTES_PER_PIXEL);
    }
}
```

[6] Como el código es bastante denso en apartados de configuración no se han añadido fragmentos, está disponible en */src/io/framebuffer.c*

[7] Puede variar según la configuración del framebuffer

La lectura, es igual de sencillo que escribir. La operación es útil para aplicar filtros o capturar la pantalla. Para lograrlo únicamente se debe calcular el puntero y acceder a él como se describe en el siguiente fragmento:

```
color_24* read_pixel(uint32_t x, uint32_t y, color_24* pix) {
    if (x < fbinfo.width && y < fbinfo.height) {
        uint8_t* location = fbinfo.buf + y*fbinfo.pitch + x*BYTES_PER_PIXEL;
        return location;
    }
}
```

Estos dos métodos sirven de base para el desarrollo de un sistema gráfico de vistas con una interfaz gráfica simple como se explica en la sección 2.9. La figura 2.9 muestra el Log de arranque del SO en la consola a partir de un framebuffer de 640x480 píxeles con un sistema de vistas creado con éste módulo.

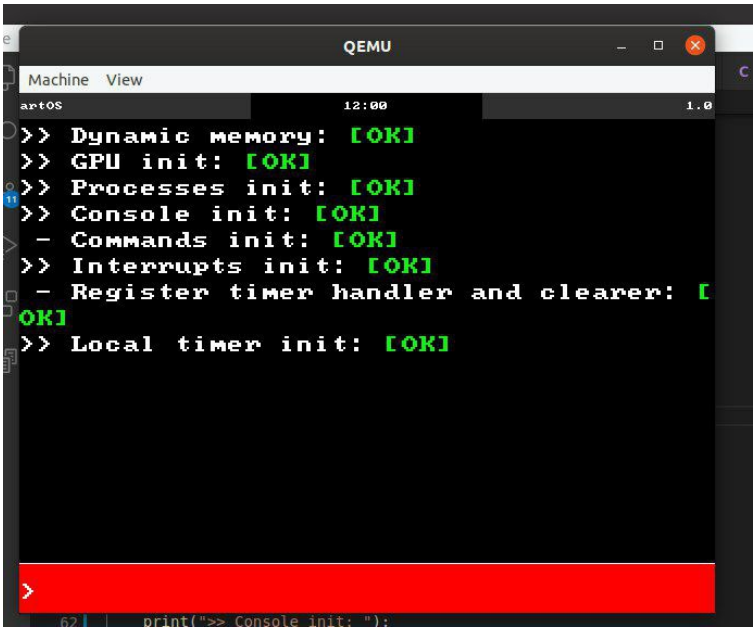


Figura 2.9: Log de arranque del SO en el sistema de vistas desarrollado

2.4.4. stdio.h

La idea de este módulo es concentrar todos los métodos de entrada y salida en una única librería estándar para futuros programadores. Por el momento agrupa la UART y la consola de la interfaz gráfica que se ha desarrollado a partir del módulo de GPU.

Incluye varias funciones de impresión, todas redirigen los datos a la UART y a la consola pero alguna tiene funciones añadidas como el salto de linea o enriquecimiento. A modo de ejemplo, la siguiente rutina muestra la implementación para imprimir por pantalla.

```
void print(char* text) {
    uart_puts(text);
    console_putStr(text, NULL, NULL);
}
```

Respecto a la entrada de datos directa del teclado, el único método que se ha desarrollado es a través de la UART. La rutina de lectura que se muestra a continuación explica el procesamiento de una línea entera (hasta encontrar el carácter de final de línea o por tamaño máximo del *buffer*).

```
1 void readLn(char* out, int max) {
2     int size = 0;
3     char c;
4     while ((c = uart_recv()) != '\n' && size < max) {
5         out[size++] = c;
6     }
7 }
```

2.5. Interrupciones

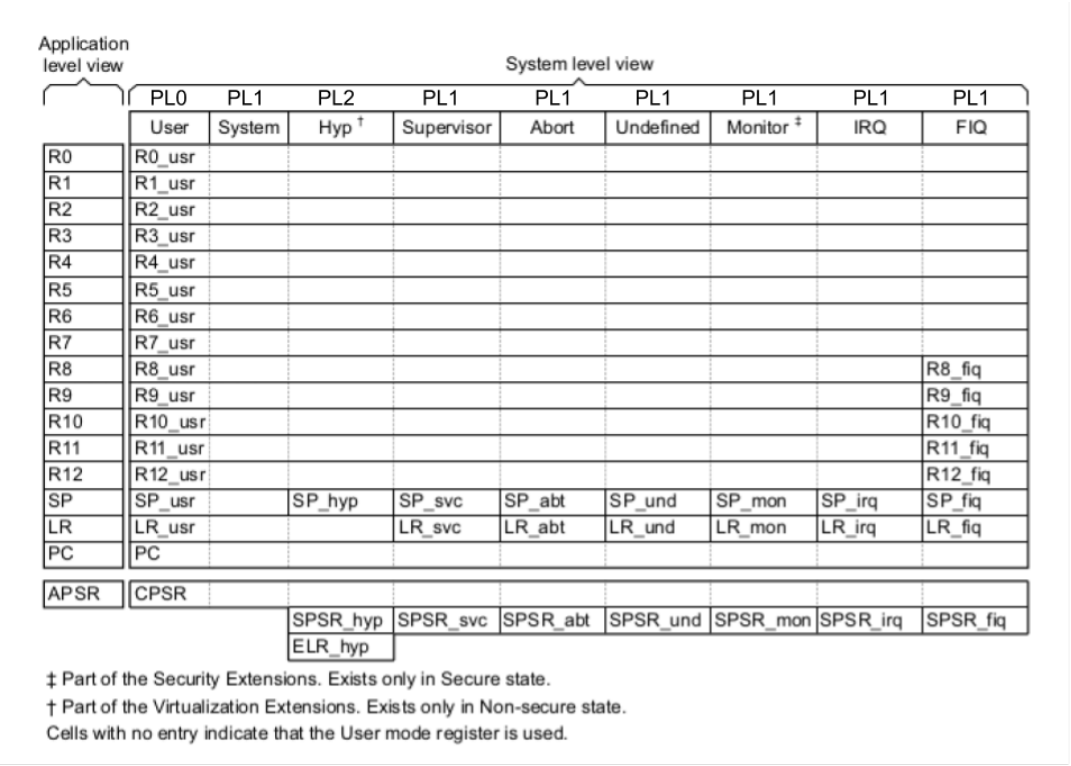
En cualquier tipo de arquitectura moderna se da soporte para que el procesador pueda tratar *excepciones*, aquí hablaremos del soporte que da ARM. Las excepciones son eventos que obligan a la CPU a dejar lo que sea que esté haciendo para tratarlos, algunos ejemplos de estos eventos son:

- Las interrupciones.
- Intentar hacer una división entre cero (ejemplo 5/0).
- Acceder a una zona de memoria prohibida.
- Intentar ejecutar una instrucción que no está definida en el repertorio de instrucciones.

Antes de continuar, debemos hacer un inciso en el diseño de ejecución de la arquitectura que estamos usando. Esta arquitectura da soporte para ejecutar en varios **modos**, estos modos de ejecución se diferencian en la cantidad de **permisos** que tienen así como en el número de registros que pueden usar. Los permisos se enumeran como PL0, PL1 y PL2 siendo PL0 el que menos privilegios tiene y PL2 el que más (PL0 y PL1 tienen privilegios con seguridad y sin seguridad[8]). Los modos de ejecución son User, System, Hypervisor[9], Supervisor, Abort, Undefined, Monitor, IRQ y FIQ.

La figura 2.10 siguiente muestra, a groso modo, diferencias fundamentales entre modos, se pueden observar en la primera columna a la izquierda los registros disponibles para el desarrollador. Cada columna representa qué registros poseen cada uno de los modos (si el campo está vacío significa que el registro es el mismo que el de la columna **User**). Todos los modos, salvo el System, poseen registros SPSR, estos registros sirven para guardar el estado del programa (Valor del registro CPSR) tal y como se explicará más adelante, también, desde el modo Hypervisor en adelante, cada uno de los modos posee su propio registro de pila con la intención de que cada uno tenga su propia pila. Se puede observar que el modo FIQ (Fast Interrupt Request) posee registros propios

[8]La extensión de seguridad no forma parte de este TFG, se deja para que futuros alumnos la complementen
[9]Contiene soporte para virtualización. Tampoco forma parte de este TFG



Application level view	System level view								
	PL0	PL1	PL2	PL1	PL1	PL1	PL1	PL1	PL1
	User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

‡ Part of the Security Extensions. Exists only in Secure state.
† Part of the Virtualization Extensions. Exists only in Non-secure state.
Cells with no entry indicate that the User mode register is used.

Figura 2.10: Diagrama de los distintos modos de un procesador

(de r8 en adelante), esto se debe a que este tipo de interrupciones están pensadas en la velocidad del tratamiento, por esa misma razón suele colocarse la rutina de tratamiento justo después de la Vector Table.

A continuación se detallan los pasos que se deben tomar para manejar una excepción.

La forma que tiene el CPU para tratar las excepciones es mediante la *vector table*. La vector table normalmente empieza[10] en la dirección 0x00000000. El contenido de la *vector table* suele ser una única instrucción de salto a la subrutina que trata la excepción correspondiente (también depende de la arquitectura, normalmente ARM hace eso), la figura 2.11 muestra qué tipo de excepción va a ser la que trate cada contenido de la vector table, hay que sumar (o restar) el offset a la dirección en la que empiece el vector.

Una vez explicado esto, podemos introducir los pasos que se toman y hay que tomar cuando ocurre una excepción.

Al ocurrir una excepción, se realizan estos pasos automáticamente

- CPSR (Current Program Status Register) en el registro auxiliar SPSR (Saved Program Status register) del modo que va a tratar la excepción.
- Se actualiza el registro CPSR para que refleje que estamos en otro modo distinto, entre otras cosas.

[10] Se puede configurar para que empiece en la 0xFFFFFFFF, para más información sobre el proceso, leer architectural manual sección B1.8.1 Exception vectors and the exception base address

Offset	Vector tables			
	PL2 only	All modes in PL1 except Monitor		
	Hyp[a]	Monitor[b]	Secure	Non-secure
0x00	Not used	Not used	Reset	Not used
0x04	Undefined Instruction, from Hyp mode	Not used	Undefined Instruction	Undefined Instruction
0x08	Hypervisor Call, from Hyp mode	Secure Monitor Call	Supervisor Call	Supervisor Call
0x0C	Prefetch Abort, from Hyp mode	Prefetch Abort	Prefetch Abort	Prefetch Abort
0x10	Data Abort, from Hyp mode	Data Abort	Data Abort	Data Abort
0x14	Hyp Trap, or Hyp mode entry	Not used	Not used	Not used
0x18	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
0x1C	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

a. Non-secure state only. Implemented only if the implementation includes the Virtualization Extensions.
b. Secure state only. Implemented only if the implementation includes the Security Extensions.

Figura 2.11: Diagrama de la Vector Table

- Se guarda la dirección en la que surgió la excepción en el registro LR del modo al que cambiamos.
- Se actualiza PC para que vaya a la posición de la vector table que contiene la instrucción de salto para tratar la excepción (se hace PC = *DIR INICIO VECTOR TABLE* +- *offset*).

Lo anterior mencionado es lo que hace la CPU automáticamente, ahora mencionaremos lo que se tiene que hacer manualmente:

- El programador debe guardar el estado del procesador en la pila del modo que está tratando la excepción.
- Tratar la excepción.
- Restaurar el estado del procesador.
- Introducir la dirección de vuelta y restaurar CPSR. La dirección de vuelta requiere que se le aplique un ajuste, la figura 2.12 muestra qué ajustes hay que tomar (la instrucción con el condicional *s* y los registros pc y r14 (lr) permite restaurar CPSR a la vez que se hace el ajuste).

La razón por la que a lo largo de esta sección se han comentado todas las características que posee la arquitectura para tratar todo tipo de excepciones es que esos mismos conceptos se aplican

Exception	Adjustment	Return instruction
SVC	0	MOVS PC, R14
Undef	0	MOVS PC, R14
Prefetch Abort	-4	SUBS PC, R14, #4
Data abort	-8	SUBS PC, R14, #8
FIQ	-4	SUBS PC, R14, #4
IRQ	-4	SUBS PC, R14, #4

Figura 2.12: Ajustes para volver de una excepción

a las interrupciones. En la siguiente sección, crearemos algo con lo que poner a prueba los temas tratados.

2.5.1. Local timer

En la sección anterior se ha detallado cómo diseñar la vector table y cómo personalizar rutinas de tratamiento. A continuación se detalla cómo aplicar esos conocimientos para tratar una interrupción IRQ. En particular, controlaremos el Local timer, un reloj del sistema que nos servirá para crear eventos periódicamente, cosa que aprovecharemos para la posterior gestión de procesos.

Como bien hemos mencionado, el timer que usaremos se denomina *local timer* aunque a veces se puede encontrar con el nombre de *generic timer*. Es un conjunto de relojes que tiene cada núcleo de la CPU, hay 2 relojes físicos, uno virtual y un contador. Cada reloj se utiliza en modos que tengan los permisos[11] adecuados:

- Uno de los relojes físicos solo se puede usar en el modo PL2.
- El otro se puede usar en los modos PL1 con seguridad y sin seguridad.
- El reloj virtual solo es accesible desde el modo PL1 sin seguridad.

Debemos mencionar que estos timers reciben su valor desde lo que se denomina *system counter*, a efectos prácticos no nos interesa aún esta parte.

Configuración

Al igual que pasa con otros componentes como la UART, para configurar el *local timer* tenemos una serie de registros mapeados en memoria además de una cierta funcionalidad dada a través de la interfaz CP15[12] del CPU.

[11] Recordamos, permisos PL0, PL1 o PL2

[12] Para conocer más acerca de esto, leer del architectural manual [4] la sección A2.9 Co-processor support

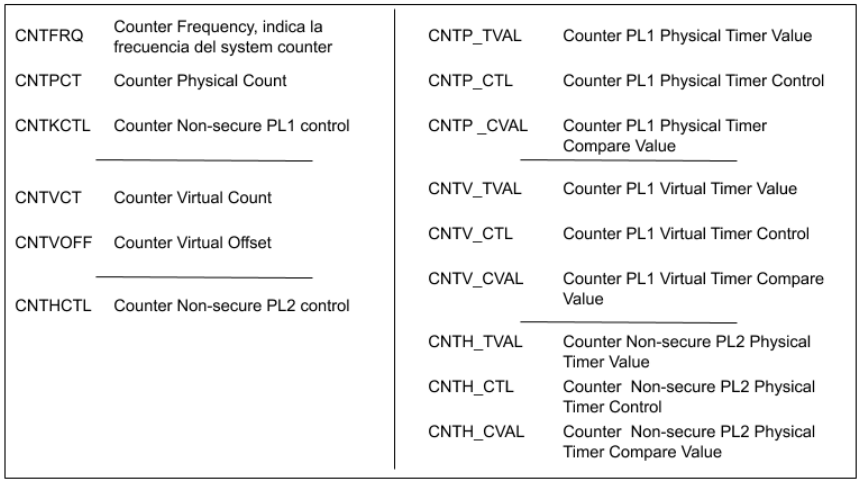


Figura 2.13: Ajustes para volver de una excepción

La dirección de memoria en la que se empieza a tratar todo lo relacionado con el *local timer* es lo que llamaremos *dirección base* en 0x40000000 estando desde el rango [*dirección base* + 0x40, *dirección base* + 0x4C] los registros de control para las interrupciones y en el rango de [*dirección base* + 60, *dirección base* + 6c] los registros para conocer de dónde viene la interrupción.

Los registros que se pueden acceder gracias a la interfaz CP15 son los mostrados en la figura 2.13, con estos registros se puede controlar cuándo salta la interrupción (CVAL), ver el valor del reloj (TVAL) y habilitar las interrupciones cuando TVAL sea igual al CVAL entre otras cosas (CTL)

Emplearemos el reloj virtual[13]. Los pasos para inicializar el *local timer* son bastante sencillos:

1. Accedes al correspondiente registro CTL para habilitar el timer.
2. Se introduce el valor deseado en el registro CVAL, ejemplo, si se quiere que haya una interrupción cada segundo, se pone en CVAL el valor de la frecuencia del *system counter* (usando CNTFRQ).
3. Se Habilitan las interrupciones en el correspondiente registro situado en *dirección base* + *offset*.

Para que el trabajo de leer y escribir en los registro CNTx_TVAL, CNTx_CTL y CNTx_CVAL sea más fácil, se ofrecen unas funciones en la librería *local_timer.c*.

[13]Sin virtualización, el contador virtual tiene los mismos valores que el físico

2.6. Memoria

Para poder utilizar la memoria de una manera ordenada, necesitamos estructurarla de manera que no haya carreras de datos al utilizar una zona de memoria ya en uso. Para ello, primero es necesario saber la cantidad de memoria total del sistema, la cual obtendremos utilizando los **atags**.

2.6.1. Atags

Los atags son el método que tiene la Raspberry Pi 2 de pasar información sobre el hardware a la imagen del SO. Estos atags se calculan en el proceso de arranque, y son pasados al kernel como una lista, la cual empieza en la posición de memoria 0x100, además de pasarse como parámetro al kernel usando el registro r2.

Dado que decidimos centrarnos más en el emulador que en el hardware, como se en el Capítulo 4.2, no se desarrolló más el archivo atags.c, el cual contiene el código para devolver la cantidad total de memoria disponible. Sin embargo, en *atags.h*, se observan todas las estructuras que manejan los atags [16].

Cuando ejecutamos el código en el emulador, no hay un proceso de arranque, por lo que los atags no contienen información. Por esto, devolvemos que la cantidad de memoria total son 256 MB, parámetro que decidimos nosotros a la hora de iniciar el emulador en el Makefile. Este parámetro se seleccionó así debido a que 256 MB nos pareció suficiente memoria para la extensión actual del proyecto. Una memoria mayor provocaría un tiempo de inicialización mayor sin ser necesario.

2.6.2. Organización de la memoria

Una vez obtenida la cantidad total de memoria, necesitamos dividirla en páginas para poder manejarla de una manera sencilla. Hemos decidido utilizar páginas de 4 KiB, ya que son lo suficientemente grandes para almacenar los datos que necesitamos, pero sin perder una gran cantidad de espacio al reservar una página. De esta manera obtenemos un total de 65536 páginas.

Esta organización de memoria está inspirada en el tutorial de Jake Sandler[17]. Teniendo la memoria dividida en partes iguales, vemos tres secciones diferenciadas en nuestro espacio de memoria, como vemos en la figura 2.14:

- **Sección del kernel:** esta sección consta de 280 páginas, y contiene:
  - **El propio código del Sistema Operativo:** sabemos donde termina gracias a la variable *__end* del archivo linker.ld, lo que nos ayuda a manejar la memoria sin temor a sobreescribir el código.
  - **Los metadatos de las páginas:** formateada como un array de metadatos. Estos metadatos constan de:
    - **Vaddr_mapped** para indicar la dirección de memoria virtual a la que pertenece la página. Esto se utilizará como soporte para trabajo futuro, ya que no hemos implementado memoria virtual, como explicaremos en el Capítulo 4.2.

- **Flags:** los cuales son *allocated* (indica que la página está reservada), *kernel_page* (indica que la página pertenece a la sección del kernel) y *kernel_heap_page* (indica que la página pertenece a la sección del heap).

- **Sección del heap:** consta de 256 páginas (1MiB), y será utilizada en memoria dinámica (sección 2.7).

- **Sección "libre":** contiene el resto de páginas, un total de 65000, que no estarán reservadas y serán utilizadas en otros módulos, como el de procesos o el sistema de ficheros, utilizando para ello los métodos *alloc_page()* y *free_page(pointer)* del archivo *mem.h*.

Figura 2.14: Estructura de la memoria.

Una vez que tenemos las secciones en las que vamos a dividir la memoria, queda inicializarla. En esta inicialización necesitamos:

1. Obtener cuántas páginas ocupará cada una de las secciones.

2. Recorrer el array de metadatos, poniendo los flags adecuados según la sección en la que se encuentre la página.

3. Añadir los metadatos (su dirección) de las páginas libres a una lista. Esta lista facilitará la implementación de los métodos *alloc_page()* y *free_page(pointer)*. En el caso de alloc_page(), sacaremos una dirección de la lista, modificaremos el flag *allocated*, calcularemos la dirección de la página usando la dirección de los metadatos y la devolveremos tras limpiarla (poner a cero la página entera). En el caso de free_page(pointer), se hará el proceso inverso, calculando la dirección de los metadatos utilizando la dirección de la página, cambiando *allocated* y guardando la dirección calculada en la lista.

2.7. Gestor de memoria dinámica

Con la estructura de memoria que hemos explicado hasta ahora, la mínima cantidad de memoria que podríamos reservar es una página, es decir 4 KiB. Esta cantidad de almacenamiento es demasiado grande si solo queremos reservar, por ejemplo, una array de char de tamaño arbitrario. Para solucionar esto, necesitamos un mecanismo para obtener zonas de memoria del tamaño deseado. Este mecanismo es el gestor de memoria dinámica.

Para implementarlo, necesitamos una zona de memoria de la que poder obtener segmentos del tamaño requerido, e información de control para saber qué partes están siendo utilizadas. La zona de memoria que utilizaremos es la sección del Heap del punto anterior, considerando las 256 páginas de esta un único gran bloque de memoria, el cual se utilizará como una doble lista enlazada. Por lo tanto, la información de control que utilizaremos por cada segmento de memoria es la siguiente:

- **Next**: un puntero al siguiente segmento de memoria. Será NULL si el segmento es el último de la lista.
- **Prev**: un puntero al segmento de memoria anterior. Será NULL si el segmento es el primero de la lista.
- **Is_allocated**: indica si el segmento está siendo utilizado o no.
- **Segment_size**: el tamaño del segmento, sin tener en cuenta el tamaño de la información de control asociado a él.

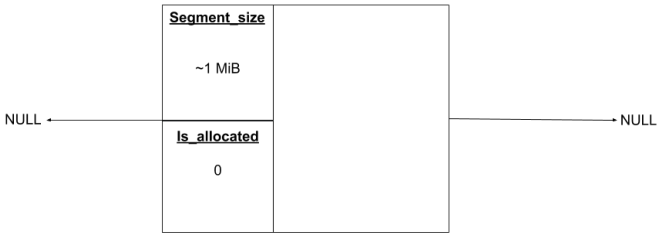
Teniendo lo anterior, ya solo nos queda inicializar la lista y crear métodos para obtener segmentos de memoria y liberarlos.

2.7.1. Heap_init(), kmalloc() y kfree()

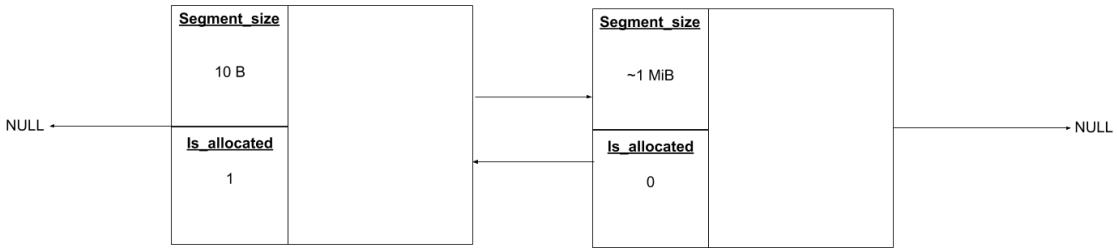
Todas estas funciones están implementadas en el archivo *berryOS/src/mem.c*. Para inicializar el heap, lo que hacemos es crear un único bloque libre de aproximadamente 1 MiB (hay que restarle el tamaño de la información de control, 16 bytes), como se observa en la figura 2.15a. Esto es lo que realiza la función heap_init().

La función kmalloc(tamaño) se encarga de devolver un puntero a una zona de memoria del tamaño específico. Para ello, busca el segmento más cercano al tamaño requerido en la lista y lo devuelve. En caso de que este tamaño sea demasiado grande (el criterio que se utiliza es que el espacio sobrante sea mayor que dos veces el tamaño de la información de control), se divide el bloque en dos partes, uno del tamaño requerido, y otro con el espacio sobrante. Este mecanismo se observa en las figuras 2.15a y 2.15b.

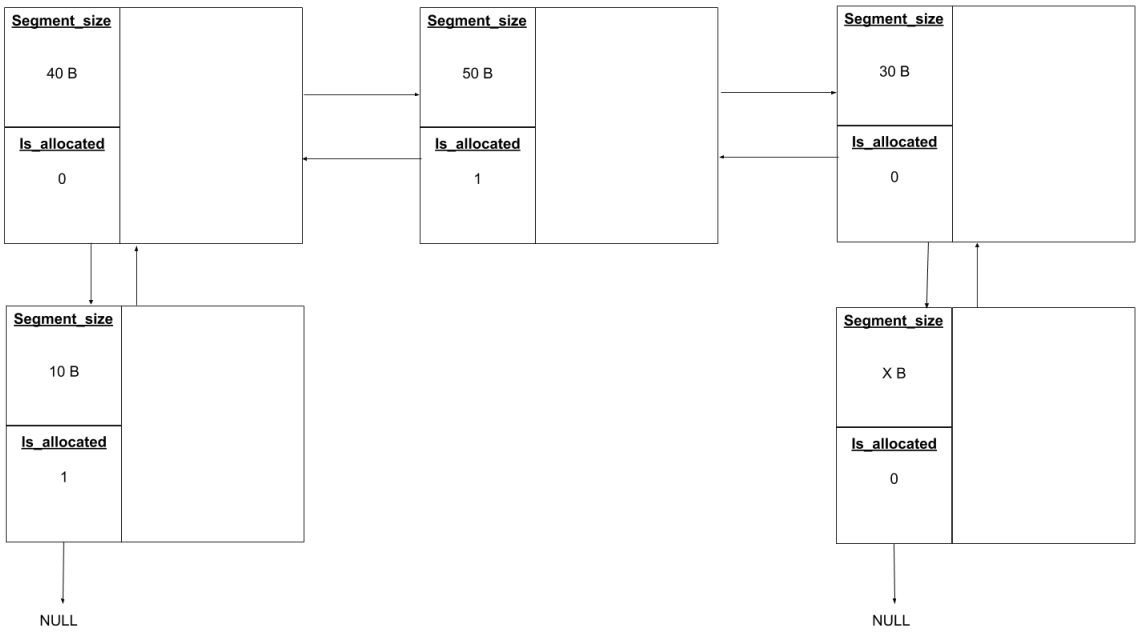
Finalmente, la función kfree(puntero), se encarga de liberar el puntero de memoria. Para ello, se pone el valor *is_allocated* a 0. Además, para *desfragmentar* la memoria, se ha incorporado un mecanismo que se encarga de *absorber* a los segmentos colindantes no utilizados, creando un bloque más grande, hasta encontrar un bloque utilizado o el inicio o el final de la lista. Un ejemplo de este mecanismo serían las figuras 2.15c y 2.15d



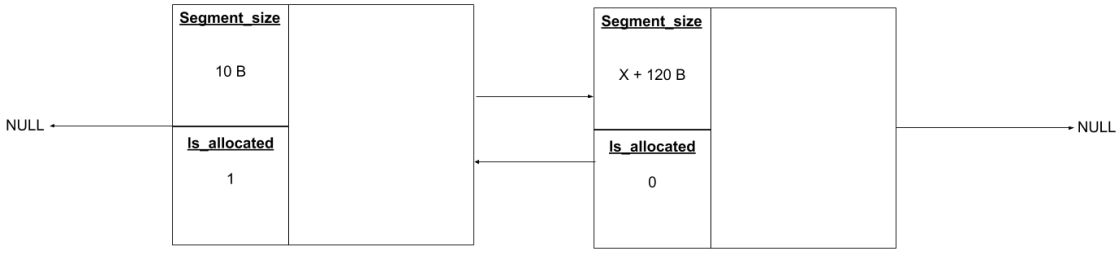
(a) Memoria tras heap_init()



(b) Tras realizar un kmalloc(10)



(c) Tras una serie de operaciones.



(d) Tras un kfree(pointer), siendo pointer el puntero al bloque de tamaño 50

Figura 2.15: Ejemplo de funcionamiento de heap_init(), kmalloc() y kfree()

2.8. Procesos

Una vez que tenemos implementados el gestor de memoria principal y el timer, tenemos lo necesario para poder implementar procesos del kernel.

Para implementar los procesos, es necesario que definamos ciertos componentes principales del módulo. El sistema operativo será el encargado de gestionar los procesos por lo que necesita una estructura para identificarlos, esa estructura es la PCB. La estructura de la PCB se encuentra en *berryOS/include/proc/pcb.h*, se denomina *process_control_block_t* y consta de los siguientes campos:

- **stack_pointer_to_saved_state**: Este campo almacenará el valor del registro sp $stack\ pointer$, este registro siempre está apuntando a la cima de pila, y la pila que usamos es de tipo *full descending*[14].
- **stack_page:** El inicio de la pagina que utilizaremos como pila de proceso. Nos servirá para implementar control sobre la cantidad de información que se introduce en la pila.
- **pid**: el identificador del proceso.
- **DEFINE_LINK(pcb)**: Esto es una directiva de la librería *berryOS/include/utils/list.h*. Con esta librería podemos crear listas de datos genéricas (independientes del tipo de datos que se le introduce). La funcionalidad de la directiva es crear punteros al siguiente y al anterior nodo denominados *nextpcb* y *prevpcb* respectivamente. Esa decisión de diseño viene dada porque crearemos una lista denominada **run_queue** que tendrá todas las PCBs asociadas a los procesos que esperan para poder ejecutarse.
- **proc_name**: Donde almacenaremos el nombre del proceso.

Merece la pena hacer notar que, usamos el término "procesos en ejecución", por lo que en este módulo también implementaremos un planificador básico, el *Round-Robin*, y daremos la posibilidad de incluir el resto de planificadores académicos como el *First Come First Serve*, *First In First Out* e incluso la posibilidad de implementar uno propio al que denominamos *OTHER*.

Ahora hablaremos sobre la inicialización de la parte de procesos. Esta inicialización se realiza en la función *process_init()* la cual hará lo siguiente:

1. Inicializamos la run_queue con la macro INITIALIZE_LIST(lista, tipo).
2. Crearemos el proceso Init. Para ello, hay que inicializar su información de control.
    - Reservaremos espacio para el nombre, un PID y la página de memoria en la que situaremos su pila.

[14] La pila va de direcciones mayores a direcciones menores

- Basándonos en la dirección de la página reservada, calcularemos la posición inicial del campo stack_pointer_saved_state como el final de la página menos el tamaño de la estructura *proc_saved_state_t*. Esta estructura, que contiene cada uno de los registros, se utilizará para inicializar el proceso cuando entre en ejecución por primera vez.
- Inicializaremos los valores de la estructura anterior. Los valores de r0-r12 los inicializaremos con basura, dado que el proceso no los utilizará. El valor de pc será la dirección de *init_function()*, que contiene el código de Init. El de *lr* será la dirección *reap()*, que se encargará de borrar el proceso cuando termine su ejecución (sección 2.8.4). Finalmente, en el *cpsr* pondremos el valor que indica que el modo de ejecución es supervisor.

3. Añadimos el proceso Init a la run_queue.
4. Elegiremos como scheduler principal el que tenemos asignado por defecto, basado en Round-Robin.

2.8.1. Secuencia de ejecución de llamada al scheduler

El timer que creamos como ejemplo de uso de las interrupciones en la sección 2.5.1 lo emplearemos para crear el planificador de nuestro módulo de procesos. Este timer tendrá un cuanto definido con una directiva denominada *QUANTUM* (reside dentro del archivo *berryOS/include/-local_timer.h*).

La secuencia es la siguiente:

1. Cuando salta el timer, se genera una interrupción de tipo IRQ, esa interrupción será tratada por una rutina personalizada (tal y como se menciona en la sección 2.5 de interrupciones). Esa rutina reside dentro del archivo *berryOS/arch/ARMv7/interrupt_init.S* y se denomina *irq_s_handler*.
2. Al tratar la interrupción comprobaremos quién la ha generado, en ese caso es el scheduler por lo que llamaremos a su correspondiente rutina de tratamiento, esta rutina se denomina *schedule* y reside dentro de *berryOS/src/proc/pcb.c*. Lo que se hará en esta rutina es comprobar qué scheduler queremos utilizar y aplicar su algoritmo correspondiente.
3. Una vez se está aplicando la labor de planificación, se realizará el cambio de contexto para que el siguiente proceso pueda proceder a su ejecución. El cambio de contexto diseñado en este TFG es un proceso complejo y el cual se explicará en la sección 2.8.2.
4. Al acabar de realizar el cambio de contexto, retornamos a la rutina de tratamiento de la interrupción del apartado 1 y damos paso a que empiece a ejecutar el proceso actual por el lugar en donde se quedó la última vez que ejecutó o que empiece por primera vez.

2.8.2. Cambio de contexto

La esencia del cambio de contexto es que podamos guardar el estado de un proceso en memoria y cargar el estado del proceso que pasa a ejecutar de la memoria, esto lo hacemos gracias al

planificador que hemos implementado. Puesto que no tenemos a nuestra disposición la memoria virtual, tuvimos que implementar una versión que fuese compatible sin ella. El código de cómo se realiza el cambio de contexto viene dado en el archivo *berryOS/src/proc/context.S*, todo lo que haremos, tendrá que ser escrito en ensamblador.

Dentro del archivo mencionado podemos encontrar tres subrutinas clave: *yield_to_next_process*, *load_process* y *switch_process_context*. Sin entrar en mucho detalle, cada una de ellas tiene un papel clave, la primera sirve para cargar el siguiente proceso una vez el actual ha terminado de ejecutar definitivamente (proceso explicado en la sección 2.8.4), la siguiente sirve para cargar el primer proceso de todos, Init, y la última para cambiar de un proceso en ejecución por el siguiente.

Las tres subrutinas siguen un mismo patrón, por lo que vamos a hablar primero de cómo cargamos un hipotético proceso de memoria. Recomendamos encarecidamente que se tenga delante la subrutina *switch_process_context*, lo que vamos a explicar tiene lugar en la segunda mitad del código de esa subrutina.

Cargar de memoria y guardar en memoria son procesos, en sí mismos, sencillos no obstante vamos a documentar esta gestión por pasos ya que es fácil perderse. Hemos decidido que cuando un proceso se le quita de la CPU, todo su estado será guardado en su pila de ejecución, esa pila la podemos encontrar con el nombre de *stack_pointer_to_saved_state* dentro de su estructura de PCB. Como bien sabemos, las pilas son *full descending* lo que significa que la cabeza de la pila está apuntando al primer elemento del estado del proceso y el resto viene después, la estructura[15] es la siguiente:

```
typedef struct {
    uint32_t cpsr; //(Saved Process State Register)
    uint32_t* lr; //pointer to return address
    uint32_t* pc;
    uint32_t r12;
    uint32_t r11;
    uint32_t r10;
    uint32_t r0;
    uint32_t r1;
    uint32_t r2;
    uint32_t r3;
    uint32_t r4;
    uint32_t r5;
    uint32_t r6;
    uint32_t r7;
    uint32_t r8;
    uint32_t r9;
} proc_saved_state_t;
```

Después de r9 están datos importantes que no queremos corromper, por lo que lo único que realizaremos es una extracción ordenada de los registros y procederemos a guardarlos en posiciones adecuadas de la pila de interrupciones, ahora mismo no importa en cuáles, luego se explicará con más detalle.

[15] Esta estructura se puede encontrar en *berryOS/include/proc/pcb.h*

1. Guardaremos el registro cpsr en el spsr, recordamos que durante la ejecución de la subrutina, seguimos en el estado IRQ, de esta lo que garantizamos es una recarga automática del registro cpsr correcto

2. Después guardamos el registro lr en una variable global que hemos denominado *__process_lr* y la cual emplearemos en el tramo final de la rutina *irq_s_handler*

3. Ahora procederemos a obtener los registro pc, r12, r11, y r10 y guardarlos, como ya hemos mencionado, en posiciones específicas de la pila de interrupciones

4. El paso final es obtener el resto de registros y realizar el mismo proceso que en el punto anterior

Una vez hecho esto, marcaremos un flag denominado *__scheduler_finished* y guardaremos el valor actual de *stack_pointer_to_saved_state* en una variable global llamada *__process_sp* para su posterior uso al final de la subrutina en ensamblador de tratamiento de interrupciones.

Lo que se hará para cargar el proceso consiste en el siguiente fragmento de código sacado de la subrutina *irq_s_handler*.

```
bl irq_c_handler
pop {r0-r12, lr}

    push {r0}
    ldr r0, =__scheduler_finished
    ldr r0, [r0]
    cmp r0, #0x1 //Have the scheduler interrupt been treated?
    bne normal_irq_execution
        pop {r0} //we recover the correct value of r0
        push {r0, lr} //we store r0 with lr into irq stack
        ldr r0, =__process_lr
        /* we load where we will return from the function
           where the process was interrupted */
        ldr r0, [r0]
        push {r0} //we make sure lr contains a pointer to that region of memory

        ldr r0, =__stack_memory
        str sp, [r0]
        mov sp, #IRQ_STACK

        mrs r0, spsr
        msr cpsr_cxsf, r0 //we change cpsr to the corresponding value of cpsr
            of the new process
        /* we load the correct value of r0 and we return to where our new
           process were interrupted */
           //we change execution mode so does the stack and lr
        ldr r0, =__process_sp //we update sp value
        ldr sp, [r0]
        ldr r0, =__stack_memory
        ldr r0, [r0]
        ldmfd r0!, {lr}
        ldmfd r0, {r0, pc}

normal_irq_execution:
    pop {r0}
    //rfeia sp! //we do the inverse operation of srsdb
```

```
36      subs pc, lr, #4
```

La finalidad de todo este código es obtener en la pila un resultado como el de la figura 2.16, y así poder introducir esos valores en los registros correspondientes. La variable *__stack_memory* nos sirve para saber dónde empiezan esas zonas de memoria especiales de la pila irq que mencionamos antes, para cuando el código llega a esta parte, esa variable ya ha cumplido su finalidad, por lo que la emplearemos como variable temporal para no perder la dirección de memoria que apunta a la figura 2.16.

Como podemos ver al final del código, recuperaremos la pila del proceso que va a entrar gracias a *__proces_sp* y recuperaremos el resto de registros gracias a la variable mencionada en el párrafo anterior.

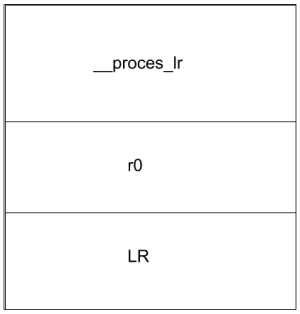


Figura 2.16: Pila irq al final de la subrutina irq_s_handler

Ahora explicaremos el proceso contrario, cómo guardar en la pila del proceso su estado. Recomendamos encarecidamente que se tenga delante el código de la primera mitad de la subrutina *switch_process_context*.

Antes de explicar nada, introduciremos el que será el paso fundamental de todo este proceso.

```
1   irq_s_handler:
2   /*
3       It is necessary to switch to supervisor mode and store some registers
4       into it's stack for having support for nested exceptions
5       */
6       push {r0-r12, lr}
7       ldr r12, =__stack_memory
8       str sp, [r12]
9       ldr r0, =__scheduler_finished
10      mov r12, #0x0
11      str r12, [r0]
12      bl irq_c_handler
```

Ese paso fundamenta está en este código, consiste en guardar en la variable global *__stack_memory*

la dirección a la que apunta el registro sp de interrupciones una vez guardado el estado del proceso que está ejecutando actualmente para poder tratar la interrupción. Como se puede notar, este paso es fundamental ya que gran parte del trabajo para obtener el estado del proceso en ejecución se hace por defecto al tratar cualquier tipo de interrupción.

Ahora es explicaremos qué hacemos en la primera mitad de la subrutina *switch_process_context*.

1. Obtenemos el valor de *stack_pointer_to_saved_state* de el parámetro adecuado de la función.
2. Una vez obtenido ese valor, lo que necesitamos hacer es recuperar el registro sp del proceso. Recordamos que al cambiar a el modo irq, se cambian ciertos registros, entre ellos el sp, por lo que necesitaremos volver al modo de ejecución en el que está actualmente el proceso para obtener dicho registro. También necesitaremos recuperar el registro lr del proceso ya que también le ocurre lo mismo que al registro sp.
3. Ahora obtenemos en orden el valor del estado del procesador que ha sido guardado en la pila de interrupciones y al cual podemos acceder gracias a la variable *__stack_memory*.
4. El orden en el que se extrae y se guardan los registros es el siguiente: r0-r9 primero y luego r10, r11, r12, lr[16] y cpsr. Estos registros serán guardados cumpliendo la estructura descrita en el código introducido al principio de esta sección.

Una vez explicado esto, el resto de subrutinas siguen los mismos patrones por lo que ya podemos pasar a hablar de cómo crear un proceso nuevo.

2.8.3. Crear un nuevo proceso

Para crear un proceso nuevo, emplearemos la subrutina siguiente subrutina

```
1 void create_kernel_thread(kthread_function_f thread_func, char * name, int
      name_size);
```

Los pasos para crear un nuevo proceso son los mismos que para crear el proceso Init, pero con dos diferencias. La primera es que el nombre del proceso nos lo pasarán por parámetro, por lo que tendremos que comprobar que no supere la máxima longitud de nombres. La segunda es que en pc guardaremos el puntero a la función que nos pasan por parámetro, que contendrá el código a ejecutar por el proceso.

2.8.4. Finalizar un proceso: reap()

Recordemos que los procesos se realizan en una subrutina. Debido a esto, cuando finalice la subrutina, se sobrescribirá el valor de pc con lr. Gracias a esto, y a que al inicializar el proceso, guardamos en lr el puntero a la función reap(), cuando finalice la ejecución del proceso, se ejecutará reap().

[16] Este registro no es el lr del proceso, es el lr que se guardó en la pila al iniciar la rutina de tratamiento de interrupción, por consiguiente, es el lugar en el que se ha interrumpido al proceso, su pc

Por tanto, reap() contendrá el código para eliminar un proceso finalizado. Para ello tendremos que:

1. Sacar de la run_queue el siguiente proceso a ejecutar.
2. Liberar la página que estaba usando el proceso.
3. Liberar la estructura con la información de control del proceso.
4. Cambiar al contexto del nuevo proceso con la función yield_to_next_process()

2.8.5. Interfaz para registrar nuevos scheduler

La interfaz para registrar planificadores es muy sencilla, primero necesitaremos unas estructuras de control y luego subrutinas que comprueben el estado de esas estructuras.

```
typedef enum {
    FIFO = 0,
    RR = 1,
    FCFS = 2,
    OTHER = 3
} sched_type_t;

typedef struct {
    kscheduling_function_f sched_function;
    char registered;
} sched_f_control_t;

typedef struct  {
    kscheduling_function_f by_default;
    sched_f_control_t schedulers[MAX_SCHEDULERS];
    sched_type_t using;
} sched_control_t;
```

sched_control_t contendrá los planificadores que podemos emplear (máximo 4), qué planificador estamos empleando (variable *using*) y para cada planificador, su algoritmo asociado y si está registrado o no.

Para registrar un planificador lo único que tendremos que hacer es llamar a la subrutina

```
int register_scheduler_policy(kscheduling_function_f sched_func, sched_type_t
    policy_type);
```

con el algoritmo que queremos que implemente y el tipo de planificador que será, esta subrutina nos devolverá si la operación se ha podido realizar (0) o si ha habido algún error (-1).

También podemos borrar un planificador usando la subrutina

```
void unregister_scheduler_policy(sched_type_t policy_type);
```

la cual siempre tendrá efecto y podemos cambiar el planificador que vamos a utilizar con la subrutina

```
int change_scheduling_policy(sched_type_t policy_type);
```

la cual nos devolverá si se ha podido cambiar (0) o si no ha sido posible el cambio (-1).

2.8.6. Funciones de impresión

Estas funciones fueron creadas con motivos de depuración, o para ser utilizadas en la creación de comandos.

print_processes()

Imprime por la UART el contenido de la *run_queue*, además del número de elementos contenidos. La implementación consiste básicamente en imprimir el tamaño de la lista, y de recorrer esta imprimiendo el nombre del proceso y su PID.

console_print_processes()

Esta función tiene la misma función que la de arriba, con la diferencia de que la impresión la realiza por consola en vez de por la UART. Además, también imprime el nombre y PID del proceso que se encuentra actualmente en ejecución, cuya referencia se encuentra almacenada en la variable global *current_process*. Para más información relacionada con la consola y la interfaz de usuario, ir a la sección 2.9.

Esta función se utilizará en ps_function(), el cual es el método trigger del comando ps descrito en el capítulo 3.5.2. El proceso a seguir para registrar comandos será explicado en la sección 2.9.3.

2.9. Interfaz gráfica

Una vez que se ha implementado el módulo de GPU explicado en la sección 2.4.3 se puede implementar una interfaz gráfica. En ocasiones, se infravalora este apartado, al menos cuando se trata de implementarlo a bajo nivel, pero sin duda es un tema interesante que rasca varias ramas de la ingeniería, entre ellas estructuras de datos, algoritmos eficientes y una alta relación con las matemáticas más puras.

2.9.1. Sistema de vistas

Diseño

El sistema está basado en el diseño de vistas que maneja Android [3] con la estructura de clases como la de la figura 2.17. Todo el entramado se genera a partir de dos abstracciones:

1. Una **vista** o *view*[2] es una clase abstracta java que tiene la jurisdicción de un conjunto rectangular de píxeles.

2. Un **grupo de vistas** o *viewgroup* [1] es también una clase abstracta que extiende de vista y tiene un atributo lista de hijos o *children* que pueden ser de tipo vista o grupo de vistas. Igual que la vistas, tienen el control de un grupo de píxeles pero que en este caso reparte entre todos sus hijos, esto se conoce como disposición o *layout*.

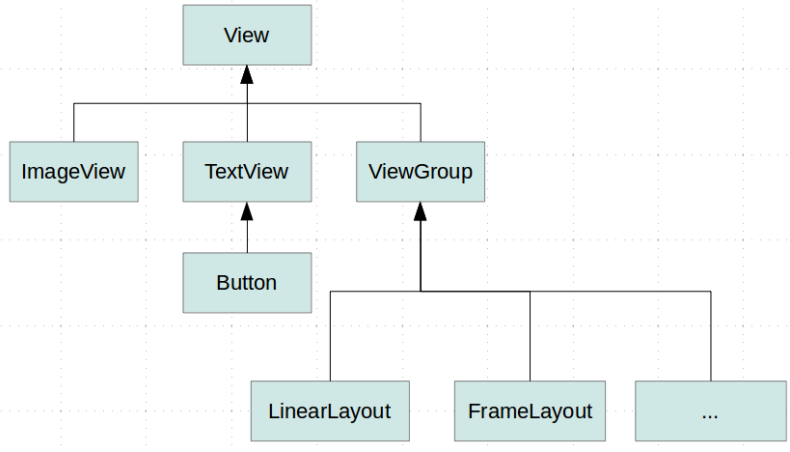


Figura 2.17: Jerarquía de clases del sistema de vistas en Android.

Implementación

C no es un lenguaje de programación orientada a objetos, por lo que las técnicas de herencia y polimorfismo que tiene java no están disponibles. Aún siendo un reto complejo, se ha conseguido un sistema semejante con el uso de punteros genéricos (void *) para conseguir el polimorfismo así como la abstracción y el paso de funciones por parámetros, típico de la programación declarativa, para la herencia.

Vista

```
typedef struct VIEW {
    int width;
    int height;
    int x;
    int y;
    int fontSize;
    color_24 bgColor;
    color_24 textColor;

    char* text;
    TEXT_ALIGN textAlign;
    int textLines;
    int textOverflow;
    int padding;
} VIEW;
```

Código 2.16: Representación de una vista o view.

Como se puede ver en el *struct* de una vista, tiene varias funcionalidades extra, pero la base, como ya se explicó en el diseño, se centra en la gestión de los píxeles designados. Su área está definida por el rectángulo formado por el valor de la altura y anchura en la posición indicada por los atributos x e y.

```
void draw(VIEW* v) {
    int width = v->x + v->width;
```

```
3       int height = v->y + v->height;
4
5       for (int i = v->x; i < width; i++) {
6           for (int j = v->y; j < height; j++) {
7               write_pixel(i, j, &v->bgColor);
8           }
9       }
```

Código 2.17: Renderización de una vista.

En el fragmento del código 2.17 se muestra cómo una vista renderiza sus píxeles asignando el color que tiene de fondo, para ello se hace uso del módulo de la GPU.

Grupo de vistas

```
1   typedef enum vtype {
2       TYPE_VIEW,
3       TYPE_VIEW_GROUP
4   } vtype;
5
6   typedef struct VIEW_OR_GROUP {
7       void* child;
8       vtype type;
9   } VIEW_OR_GROUP;
10
11  typedef struct VIEW_GROUP {
12      VIEW view;
13      VIEW_OR_GROUP_list_t children;
14      void (*layout)(void*);
15      int dirty;
16  } VIEW_GROUP;
```

Código 2.18: Definición de grupo de vistas.

Para los grupos de vistas se presentan los retos ya comentados, conseguir la herencia, el polimorfismo y la abstracción.

Con respecto al *polimorfismo*, se usa un estructurado con un puntero genérico (void *) acompañado de un enumerado que indica de qué clase se trata (línea 1-9 y 13 de 2.18).

Un grupo de vistas *extiende* de una vista, para conseguir esto, se encapsula una vista dentro de la definición (línea 12 del código 2.18).

Para conseguir la técnica de la *abstracción*, es decir, varios tipos de grupos de vistas que tengan distinta forma de distribuir las vistas, se crea un atributo variable función (línea 14 de 2.18) que viene a ser un método de una clase abstracta que cabe implementar[17].

```
1   void layoutGroup(VIEW_GROUP* vg) {
2       vg->layout(vg);
3       VIEW_OR_GROUP* node = start_iterate_VIEW_OR_GROUP_list(&vg->children);
4       while (has_next_VIEW_OR_GROUP_list(&vg->children, node)) {
5           node = next_VIEW_OR_GROUP_list(node);
6           if (node->type == TYPE_VIEW_GROUP) {
7               layoutGroup(node->child);
```

[17] Se ha creado un módulo con las implementaciones más famosas de Android en /src/ui/layouts.c, https://github.com/dacya/tfg1920-raspiOS/blob/master/berryOS/src/ui/layouts.c

```
8             }
9         }
10    }
```

Código 2.19: Cálculo de la distribución de píxeles.

La característica de un grupo de vistas, es que gestiona el conjunto de píxeles que tiene asignado entre sus hijos. La forma de distribución la describe el método *layout* que debe asignar el tamaño y la posición del área que le toca a cada uno (línea 2 de 2.19). Una vez calculado, se debe llamar a todos sus hijos a que hagan lo mismo si también son grupos de vistas, de esta forma se consigue generar un árbol de vistas (línea 4 de 2.19).

```
1   void drawGroup(VIEW_GROUP* vg) {
2       draw(&vg->view);
3       if (vg->dirty)
4           vg->layout(vg);
5       VIEW_OR_GROUP* node = start_iterate_VIEW_OR_GROUP_list(&vg->children);
6       while (has_next_VIEW_OR_GROUP_list(&vg->children, node)) {
7           node = next_VIEW_OR_GROUP_list(node);
8           if (node->type == TYPE_VIEW_GROUP) {
9               adjustGroupRelative(vg, node->child);
10              drawGroup(node->child);
11          } else if (node->type == TYPE_VIEW) {
12              adjustGroupRelative(vg, node->child);
13              draw(node->child);
14          }
15      }
16      vg->dirty = 0;
17  }
```

Código 2.20: Dibujar un grupo de vistas.

Dibujar o renderizar un grupo de vistas se trata de una función recursiva, primero se dibuja a sí mismo y luego realiza la llamada pertinente a cada uno de sus hijos dependiendo de su tipo (fragmento 2.20).

Es interesante la importancia del orden de llamadas, en el ejemplo explicado se dibuja primero el padre y posteriormente los hijos en orden *FIFO*, es decir, el padre queda en el fondo y encima están dibujados los hijos. Este orden podría ser distinto y por tanto obtener resultados visuales distintos, a esto se le suele conocer como índice z o *z-index*.

Se ha añadido el *flag dirty* para gestionar cuando no se requiere *redibujar* al completo la jerarquía y así se gana eficiencia podando el árbol. Éste se marca en las funciones de edición del grupo (cambios de color, layout o hijos) y una vez realizado el *redibujado* se desmarca.

Ejemplo de uso

```
1   VIEW_GROUP consoleView;
2   VIEW lineSeparator;
3   VIEW textInput;
4   VIEW_GROUP display;
5
6   VIEW_GROUP statusBarView;
7   VIEW brand;
```

```
8  VIEW time;
9  VIEW version;
```

Código 2.21: Definición de vistas para la consola.

Al modelar una interfaz, lo primero es instanciar las vistas de forma estática ya que es importante que se mantengan en memoria durante todo el ciclo de vida por las llamadas de *redibujo* (fragmento 2.21).

```
1  addView(&statusBarView, &brand); // nombre del SO
2  addView(&statusBarView, &time); // reloj
3  addView(&statusBarView, &version); // version
4
5  addViewGroup(&consoleView, &display); // fondo negro
6  addView(&consoleView, &lineSeparator); // linea blanca separadora
7  addView(&consoleView, &textInput); // input
```

Código 2.22: Creación del árbol de vistas del sistema operativo.

Una vez configuradas todas las vistas, se crea el árbol añadiendo una dentro de otra como se muestra en el fragmento de código abstracto 2.22. Para cada elemento añadido se vuelve a calcular y dibujar la disposición. El resultado final se puede visualizar en la figura 2.18.

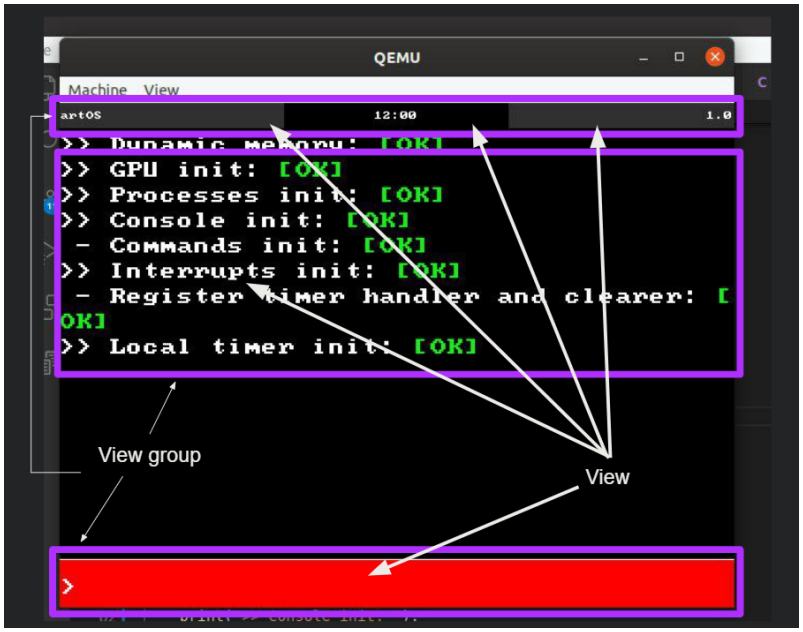


Figura 2.18: Consola diseñada con el sistema de vistas.

2.9.2. Consola

La consola es un módulo que tiene un proceso propio que se crea al inicializarse (fragmento 2.23), su función básica (rutina 2.24) es analizar la entrada de datos, ejecutar comandos y retransmitir la entrada a través de la librería *stdio.h* explicada en la subsección 2.4.4 de Entrada / Salida.

```
1  void start_console() {
2
```

```
    [...]

    create_kernel_thread(read_proc, "uart_console_input", 19);
}
```

Código 2.23: Lanzamiento del proceso al inicializar la consola

```
void read_proc(void) {
    char* comm = kmalloc(MAX_CONSOLE_LINE_INPUT_SIZE + 2);
    char c;

    do {
        c = readChar(); // bloqueante
        switch (c) {
            case 127: // del, borrar char
                [...]
            case '\n': // fin de linea
            case '\r':
                [...] // logica de comandos
            default: // input
                [...]
                print(c); // stdio.h
        }
    } while (1);
}
```

Código 2.24: Rutina del proceso de la consola

La parte gráfica de este módulo sigue la estructura explicada en 2.18. Para añadir una linea de texto o una cadena se implementan las funciones siguientes:

En la raíz del entramado de la interfaz se encuentra un grupo de vistas que contendrá grupos de vistas que representan lineas, a su vez, a esas lineas se le añadirán vistas con las cadenas de textos.

```
void console_putStr(char* str, color_24* textColor, color_24* bgColor);
void console_putLn(char* str, color_24* textColor, color_24* bgColor);
```

En el fragmento anterior, la primera rutina añade una vista a la última línea añadida, si se detecta desbordamiento realiza una llamada a la segunda función. Respecto al método para agregar una línea nueva, se añade un grupo de vistas nuevo, la línea, y dentro se crea una vista nueva con la cadena de texto, si se detecta desbordamiento realiza una llamada recursiva con la parte del texto sobrante. Adicionalmente, las dos funciones reciben por parámetro el color del texto y del fondo. Se puede visualizar la estructura que sigue en la figura siguiente:

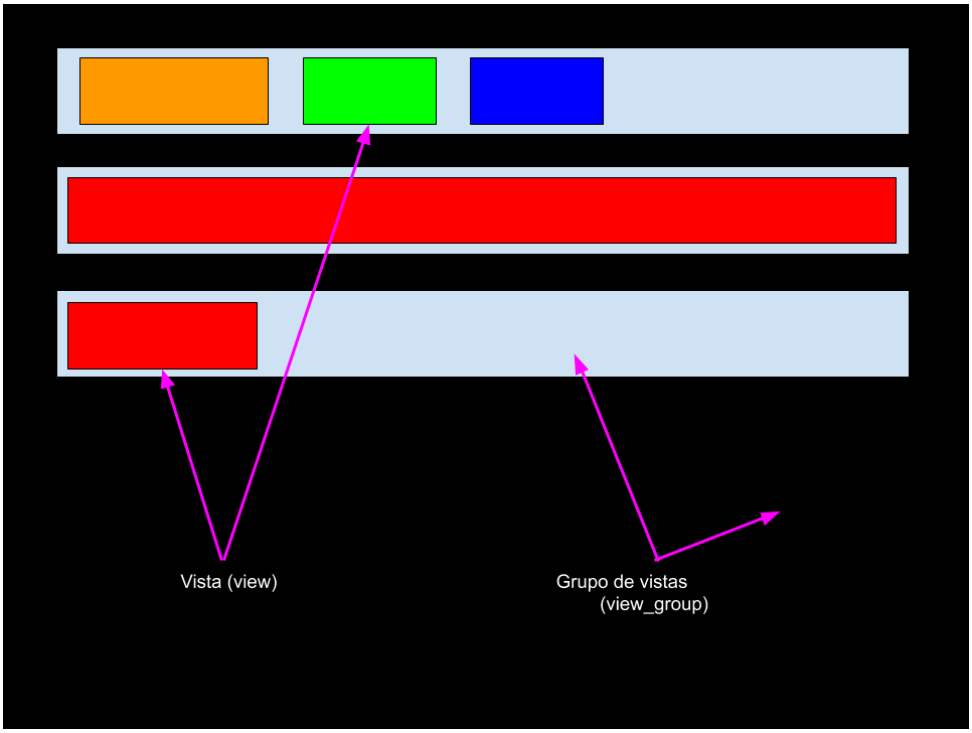


Figura 2.19: Esquema del árbol de vistas de la interfaz de la consola.

2.9.3. Comandos

Un comando está compuesto por la *palabra clave*, una *breve descripción* y el *disparador* con parámetros como una función main convencional en C.

```
typedef struct COMMAND {
    char* key;
    char* helpText;
    void (*trigger)(int argc, char** argv);
} COMMAND
```

El módulo contiene una lista dinámica con todos los comandos registrados. Se pueden añadir y eliminar a través de las funciones siguientes:

```
void regcomm(COMMAND* command);
void unregcomm(COMMAND* command);
```

Matching

Cuando la consola detecta una línea nueva, como se explica en el apartado 2.9.2, la trocea por palabras y las envía a éste módulo de comandos para que encuentre uno con la primera palabra como clave y las restantes se interpretan como parámetros.

La detección del comando se realiza recorriendo toda la lista comparando las palabras claves, sin duda, esta búsqueda se puede mejorar implementando un diccionario. Para cada comando coincidente se llamará a su disparador con la lista de parámetros (código 2.25).

```
void commatch(char* match, int argc, char** argv) {
    int not_found = 1; // flag
    comm_wrapper* commw = start_iterate_comm_wrapper_list(&commands_list);

    while (has_next_comm_wrapper_list(&commands_list, commw)) {
        commw = next_comm_wrapper_list(commw);
        if (streq(match, commw->comm->key)) { // comparacion de claves
            commw->comm->trigger(argc, argv); // llamada al disparador
            not_found = 0;
        }
    }

    if (not_found)
        enrichedPrintLn("command not found", &RED, NULL);
}
```

Código 2.25: Rutina de lanzamiento de comandos

Problema de inicialización del módulo

El orden de inicialización de módulos por el kernel influye drásticamente en los comandos. Se puede dar el caso en el que un módulo busque crear un comando y que el gestor de éstos aún no esté disponible.

Una primera aproximación sería ponerlo a la cabeza, pero al requerir de memoria dinámica no es posible por lo que se forzaría a que este módulo no pudiese tener comandos.

Por tanto, la solución alternativa ocurrida, es instanciar de forma estática un array de comandos y que al llamar a la rutina de inicialización el módulo copie todos los comandos en la lista dinámica original. De esta forma, la función de registro de comandos debe conocer cuando está instanciado el módulo y realizar las operaciones necesarias en cada situación. Se puede ver parte de la implementación en el siguiente fragmento de código:

```
COMMAND* delayedList[20];
int nDelayed = 0;
int initialized = 0;

void init_commands() {
    // proceso de inicializacion del modulo lista

    [...]

    initialized = 1; // marcar como inicializado

    for (int i = 0; i < nDelayed; i++) {
        regcomm(delayedList[i]); // volver a registrar los comandos
    }
}

void regcomm(COMMAND* comm) {
    if (initialized) {
        // memoria dinamica
        [...]
    } else {
        if (nDelayed < 20) {
```

```
23                  delayedList[nDelayed++] = comm; // array estatico
24              }
25          }
26      }
```

Ejemplo de uso

A continuación, se instancia el comando *shcomm* (del inglés *show commands*) que muestra todos los comandos registrados actualmente en el sistema. Se debe instanciar en memoria estática.

```
1   // instancia estatica del comando
2   COMMAND shcomm;
3
4   // disparador del comando
5   void shcomm_trigger(int argc, char** argv) {
6       MARK_UNUSED(argc);
7       MARK_UNUSED(argv);
8
9       int size = size_comm_wrapper_list(&commands_list);
10      comm_wrapper* comm = start_iterate_comm_wrapper_list(&commands_list);
11
12      enrichedPrintLn("Available commands", &YELLOW, NULL);
13
14      for (int i = 0; i < size;) {
15          [...]
16      }
17  }
18
19  void init_commands() {
20      [...]
21
22      // configuracion y registro
23      shcomm.key = "shcomm";
24      shcomm.helpText = "Lists all registered commands.";
25      shcomm.trigger = shcomm_trigger;
26      regcomm(&shcomm);
27  }
```

Código 2.26: Ejemplo de uso del módulo de comandos instanciando el comando *shcomm*.

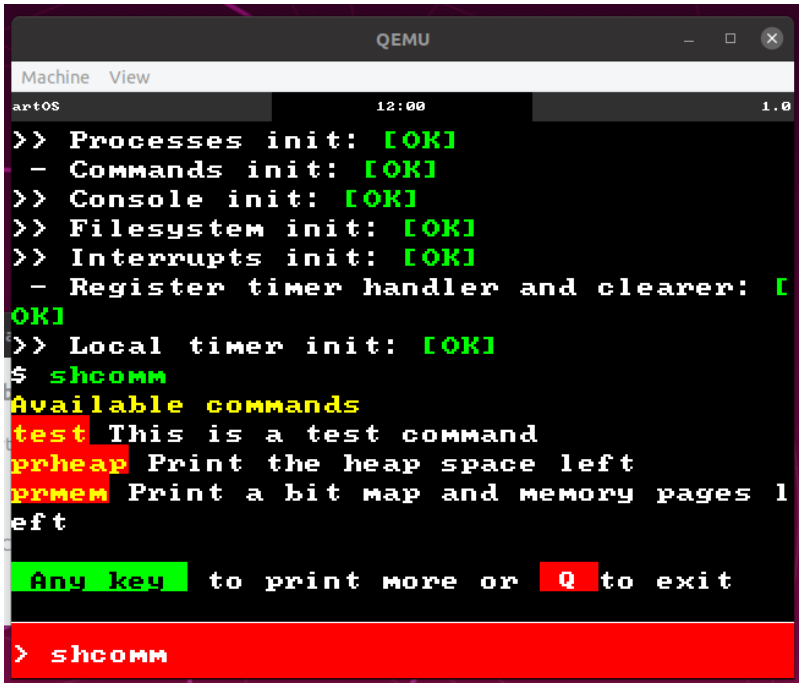


Figura 2.20: Salida del comando *shcomm*.

2.10. Cerrojos

Para implementar los locks ARMv7 [4] ofrece un sistema de marcado de direcciones de memoria según instrucciones de load y store atómicas además de una instrucción que permite desmarcar directamente (ldrex, strex y clrex).

El control del estado de esas marcas viene dado por lo que dan a conocer como un *local monitor* el cual pone ciertas restricciones para los accesos y los guardados en la memoria marcada y define dos estados *open access* y *exclusive access*.

La instrucción atómica strex es capaz de devolver si la operación ha tenido éxito (status 0) o si no se ha podido realizar (status 1) en un registro con el que poder comprobar el resultado de la operación.

La figura 2.21 muestra la tabla de transiciones para una dirección que está en un cierto estado a la que se le aplica una de las operaciones y el resultado que genera.

Initial state	Operation	Effect	Final state
Open Access	clrex	No effect	Open Access
	strex	Memory stays the same and returns status 1	Open Access
	ldrex	Load value and tag memory	Exclusive Access
	str	Normal execution	Open Access
Exclusive Access	clrex	Clear tagged memory	Open Access
	strex (on tagged addr)	Updates memory and returns status 0	Open Access
	strex (on a not tagged addr)	Can update memory and return 0 or can not do it and return 1	Open Access
	ldrex (on addr x)	Loads value and changes tag to addr x	Exclusive Access
	str	Updates memory	Can go to Exclusive acces or Open Access

Figura 2.21: Tabla de transición del *local monitor*

Con esto explicado pasamos a ver el código utilizado para implementar los locks.

```
1  lock_mutex:
2      ldr r1, =locked
3  test_lock:
4      ldrex r2, [r0]
5      cmp r2, r1 // Test if mutex is locked or unlocked
6      beq wait_lock    // If locked - wait for it to be released, from 2
7          strexne r2, r1, [r0] // Not locked, attempt to lock it
8          cmpne r2, #1 // Check if Store-Exclusive failed
9          beq test_lock // Failed - retry from 1
10             // Lock acquired
11             dmb //Required before accessing protected resource
12             mov pc, lr
13 wait_lock:
14     //WAIT_FOR_UPDATE //Take appropriate action while waiting for mutex to
           become unlocked
15     wfi
16     nop //its here but it will never be executed
17     b test_lock  //Retry from 1
18
19
20 // unlock_mutex
21 // Declare for use from C as extern void unlock_mutex(void * mutex)//
22 .global unlock_mutex
23 unlock_mutex:
24     ldr r1, =unlocked
25     dmb // Required before releasing protected resource
26     str r1, [r0] // Unlock mutex
27     //SIGNAL_UPDATE
```

```
28          mov pc, lr
```

Como podemos ver, primero realizamos una carga del valor que hay dentro del mutex con la instrucción atómica ldrex, luego comprobamos si el mutex está cogido o no, si no está cogido simplemente intentamos marcar el mutex con la instructión atómica condicional strexne y luego comprobamos el resultado de esa operación (vemos si nos ha devuelto un estado de 1 o 0), si resulta que hemos fallado la operación, volvemos a intentar coger el mutex, en caso contrario simplemente volvemos de la subrutina porque el mutex ya está cogido.

2.11. Sistema de ficheros

Nuestro sistema de ficheros está basado en el nodo indexado[14], parecido a ext2 y al cual pasaremos a denominar i-nodo o i-nodos. Todos los métodos y macros descritos en esta sección se encontraran en los archivos *berryOS/include/fs/fs.h* y *berryOS/src/fs/fs.c*.

Antes de hablar del diseño, enumeramos algunas macros que hemos desarrollado para ajustar los parámetros de nuestro sistema de ficheros, y veremos a lo largo de esta sección. Estos son:

- **MAXPAGESPERFILE**: nos indica cuántas páginas de memoria utilizaremos como máximo para almacenar la información de los archivos. Se puede modificar a conveniencia antes de ejecutar el sistema operativo.
- **MAXFILESIZE**: se calcula multiplicando el tamaño de página por la macro anterior. Por ejemplo, si utilizamos dos páginas, el tamaño máximo del archivo será de 8 KB.
- **MAXFILENAMESIZE**: el tamaño máximo del nombre del archivo. También se puede modificar a conveniencia.
- **MAXFILESPERDIR**: el número de archivos que caben en un directorio. Se calcula como el tamaño de página dividido por la estructura *dir_entry_t*, dando como resultado un máximo de 146 archivos por directorio.
- **NUM_PAGES_INODE_TABLE**: el número de páginas que se utilizarán para almacenar la tabla de i-nodos. Esta macro también se puede modificar a conveniencia.
- **NUM_INODES**: el número total de i-nodos del sistema. Se calcula dividiendo el tamaño de página entre el tamaño de la estructura *i_node_t*, y multiplicando el resultado por NUM_PAGES_INODE_TABLE. Por ejemplo, teniendo 2 páginas para la tabla de nodos, tenemos un total de 512 i-nodos.
- **NUM_INODES_PER_PAGE**: contiene cuantos i-nodos caben en cada página de memoria. Se calcula dividiendo la anterior macro por NUM_PAGES_INODE_TABLE.

Comentado lo anterior, explicaremos los nodos indexados. La información que almacenaremos en esta estructura de datos es la siguiente, contenida en la estructura *i_node_t*:

- **Size**: tiene distintos significados según el tipo del archivo. En el caso de archivos regulares, almacenará su tamaño en bytes. Sin embargo, en el caso de los directorios, contiene el número de archivos que contiene.
- **Num_pages**: contiene el número de páginas de memoria utilizadas por el i-nodo, con un máximo de MAXPAGESPERFILE páginas. Esta macro no afectará a los directorios, los cuales solo utilizarán una página, como explicaremos más adelante.
- **Type**: un bit que indica el tipo de archivo. El bit 0 indicaría fichero regular, mientras que el bit 1 indica directorio.
- **Free**: un bit que indica si el i-nodo está libre (0) o está siendo utilizado(1).
- **Pages**: las distintas páginas que está utilizando el archivo.

Debido a las limitaciones producidas por no tener reloj, no se puede añadir información como fechas de creación o modificación. Información que sí se podría añadir sería, por ejemplo, el modo (escritura o lectura).

2.11.1. Directorios

En relación a los directorios, se utilizará su página para almacenar la estructura *dir_t*, la cual contiene:

- **Num_childs**: contiene el número de hijos que tiene el directorio.
- **Child**: un array de tamaño MAXFILESPERDIR de tipo *dir_entry_t*. Los dos primeros hijos corresponderán al propio directorio, y a su nodo padre. Por otro lado, la estructura *dir_entry_t* contiene:
  - **Inode_num**: el número de i-nodo del archivo. A través de él podremos acceder a su i-nodo correspondiente.
  - **Filename**: un array de char que contiene el nombre del archivo. El tamaño máximo viene dados por MAXFILENAMESIZE.
  - **Fn_size**: el tamaño del nombre del archivo.

Los directorios utilizan solo una página de memoria, ya que 146 archivos por directorio nos parecen suficientes para la extensión actual del sistema operativo.

2.11.2. Estructura del sistema de ficheros

Nuestro sistema de ficheros está compuesto básicamente por una lista de i-nodos mapeados sobre NUM_PAGES_INODE_TABLE páginas de memoria. Decidimos no añadir estructuras como bloques, bitmaps o superbloques ya que no nos parecieron necesarios.

Utilizamos variables globales para almacenar información clave de la situación actual del sistema:

- **I_node_list_pages**: un array de tipo *i_node_t** que contiene las páginas en las que se almacenará la tabla de inodos.
- **Root_dir**: una variable de tipo textsldir_t* que contiene el directorio raíz.
- **Current_glob**: una variable de tipo textsldir_t* que contiene el directorio en el que se encuentra el usuario.
- **FreeInodes**: el número de i-nodos libres de la tabla.
- **MinFreeInode**: contiene o el último número de i-nodo que se ha reservado, o el mínimo libre en caso de que alguno haya sido liberado. Se utiliza para agilizar la búsqueda de un i-nodo libre.
- **Interface**: una variable de tipo *fs_interface* que contiene la interfaz de usuario. Esto se explicara con más detalle en la sección 2.11.9.

2.11.3. Inicialización del sistema de ficheros: fs_init()

La inicialización constará de cuatro tareas: inicializar la interfaz, registrar los comandos e inicializar la tabla de i-nodos y el directorio raíz. La inicialización de la interfaz la comentaremos en la sección 2.11.9. Para registrar los comandos, simplemente llamaremos al método register_filesystem_commands(), el cual explicaremos en la sección 2.11.8.

Para inicializar la tabla de i-nodos, necesitamos reservar una página con alloc_page() por cada una de las posiciones del array *i_node_list_pages*. Con esto, habremos reservado el espacio donde se almacenarán los i-nodos, los cuales tenemos que inicializar. Para ello, recorremos cada uno de los i-nodos señalándolos como libre, poniendo el campo *free* del i-nodo a 0. Para conseguir esto, necesitamos una función que, dado el número del i-nodo, te devuelva su posición en memoria. Esta función es get_inode(number), y la explicaremos en la próxima sección.

Inicializado ya lo anterior, solo queda crear el directorio raíz. A este le asignaremos el número de i-nodo 0, e instanciaremos el i-nodo correspondiente, asignándole que ya no está libre, que es de tipo directorio, que tiene 2 entradas, que utiliza sólo una página de memoria y reservaremos esa página de memoria.

Dado que el nodo raíz es un directorio, también tendremos que instanciar su estructura *dir_t*, que se encuentra en la página de memoria reservada. Para ello diremos que tiene dos hijos, los cuales son "." y "..", y sus números de i-nodos serán el de la raíz, es decir, 0.

Finalmente, le daremos su valor inicial a *current_glob*, que será el directorio raíz, a *free_inodes* (el cual será NUM_INODES -1) y a *minFreeInode*, el cual será 1.

2.11.4. Funciones auxiliares

Estas funciones fueron creadas para englobar código que se repite en muchas funciones, para extender las funcionalidad del sistema de ficheros, o bien utilizadas para depurar. Las explicaremos a continuación.

get_inode(number)

Como dijimos antes, devuelve el puntero asociado a un número de i-nodo, y se calcula utilizando NUM_INODES_PER_PAGE de la siguiente manera:

```
return (i\_node\_t*)(((int)i\_node\_list\_pages[num / NUM\_INODES\_PER\_PAGE])
    + (num \% NUM\_INODES\_PER\_PAGE)*sizeof(i\_node\_t));
```

getFreeInode()

Devuelve el primer i-nodo libre de la tabla. Para ello se recorren todos los i-nodos desde *minFreeInode* hasta encontrar uno libre.

fileExists(filename, first_child, father)

Comprueba si existe el archivo "filename" en el directorio "father". Para ello hay que recorrer el array *child* de father, comparando "filename" con el nombre de la estructura *dir_entry_t* contenida en cada posición del array. En el caso de que se haya encontrado se devuelve la posición en el array, y en caso contrario, -1.

Por otro lado, "firts_child" indica desde qué posición comenzaremos a recorrer el array. Esto es necesario para incluir o no en la búsqueda los nodos "." y ".." ya que, por ejemplo, no los queremos incluir al buscar un archivo para escribir (no son modificables), pero sí los queremos incluir cuando queramos cambiar el directorio actual.

calculatePath(path, filename, fnsize)

Las funciones de las secciones siguientes admiten paths absolutos y relativos. Los formatos de estos paths son los siguientes:

- **Relativos**: son nombres separados por caracteres '/'. Un ejemplo sería: "directorio/directorio/archivo"
- **Absolutos**: comienzan por "~/", seguidos de nombres separados por '/'. Un ejemplo sería "~/directorio/directorio".

Por tanto calculate_path() devolverá el último directorio (la estructura *dir_t*) referenciado en "path". También devolverá el nombre del último eslabón del path a través de "filename", y su tamaño a través de "fnsize".

Para ello, realizaremos lo siguiente:

1. Comprobaremos si el path es absoluto o relativo, comprobando el primer carácter ('~'). Si es absoluto, el directorio desde el que comenzaremos a resolver será el raíz, y si es relativo, comenzaremos desde *current_glob*.

2. Recorremos “path” carácter a carácter hasta que encontremos ’\0’, guardando los caracteres en “filename”. Si encontramos un ’/’, significa que hemos completado un nombre de directorio, y este se encuentra en “filename”. Por tanto, comprobaremos que “filename” exista en el directorio padre.

3. Si el directorio existe, obtenemos su número de i-nodo en el array *child* del padre. Con el número de i-nodo obtengo la estructura *dir_t** del directorio, y este pasa a ser el padre.

   Si no existe, devolvemos NULL indicando que la resolución ha fallado.

4. Cuando terminemos de recorrer “path”, devolveremos el *dir_t** que hemos ido calculando.

exists(path)

Una extensión de fileExists() para admitir paths. Llama a calculate_path() para obtener el último nombre de archivo de la cadena, y utiliza fileExists() para comprobar que este nombre existe en el directorio devuelto por calculate_path().

getFileSize(path)

Devuelve el tamaño de un archivo. Para ello, se resuelve “path” para obtener el directorio que contiene al archivo, y con esto obtener el i-nodo del archivo con get_inode(). Finalmente, devolvemos el campo *size* del i-nodo.

2.11.5. Funciones de manejo

En esta sección explicaremos las funciones básicas que debe tener un sistema de ficheros, como leer o escribir, por ejemplo.

getFsInterface()

Devuelve la interfaz para que el usuario la modifique. Explicaremos cómo en la sección 2.11.9.

createFile(path)

Crea un archivo en la ruta especificada. El nombre del archivo a crear será el último eslabón de la cadena.

Comenzaremos resolviendo el path para obtener el directorio que contendrá el archivo. Después, reservaremos la página donde se almacenarán sus datos y obtendremos un i-nodo con las funciones getFreeInode() y getInode().Tras esto, inicializaremos el i-nodo con la página reservada y diciendo que no está libre, tiene una página, tamaño 0 y tipo regular.

Tras lo anterior, solo queda añadir la información al directorio padre, añadiendo un hijo más a su array *child* y copiando en él el nombre del archivo, la longitud del nombre y su número de i-nodo.

createDir(path)

Crea un directorio en la ruta especificada. El nombre del archivo a crear será el último eslabón de la cadena.

Está función realiza los mismos pasos que la anterior, con la diferencia de que el tipo será directorio. Tras lo anterior queda inicializar la estructura *dir_t* del directorio a crear, añadiendo "." y ".." a su array de hijos. El número de i-nodo que añadiremos a "." será el del directorio creado, y el de ".." derá el directorio padre.

write(path, text)

Concatena al final del archivo dado por path "text" y devuelve el número de bytes escritos.

Al igual que en los otros métodos, resolveremos el path y obtendremos el i-nodo del archivo. Es necesario, antes de continuar, que el tipo del archivo sea regular.

Antes de comenzar a copiar "text" en las páginas del i-nodo, es necesario calcular el offset del archivo, así como en qué página escribiremos. Lo haremos de la siguiente manera:

```
1 char* pos = ((char*)inFile->pages[inFile->size/PAGE_SIZE]) + (inFile->size %
    PAGE_SIZE);
2 int actual_page = inFile->size/PAGE_SIZE;
```

Con el offset calculado, ya podemos copiar carácter a carácter a partir de offset. Sin embargo, es posible que al escribir, lleguemos al final de la página. Si este es el caso tendremos que hacer lo siguiente:

- Si ya hemos utilizado el máximo número de páginas permitidas (dado por MAXPAGES-PERFILE), finalizamos la transferencia y devolvemos el número de bytes escritos hasta el momento.
- Si no, si la página siguiente ya ha sido reservada, por ejemplo, porque se escribió pero se ha movido el offset hasta la página anterior (aunque actualmente no hay ningún mecanismo para mover el offset), el offset se coloca al inicio de la página reservada.
- Finalmente, si el caso no es ninguno de lo anteriores, necesitamos reservar una página nueva, en cuyo comienzo se colocará el offset.

Finalmente, actualizaremos el tamaño del archivo y devolveremos el número de bytes escritos.

read(path, bytes)

Devuelve los primeros "bytes" de contenido del archivo referenciado por "path". La función reservará el espacio necesario como un array de char, pero depende del usuario liberar la memoria una vez finalizado el uso del puntero.

Obtendremos el i-nodo resolviendo el path y comprobaremos que el fichero es de tipo regular. También es necesario obtener el mínimo entre el tamaño del archivo y el número de bytes requerido, para no leer basura, además de calcular el offset, que será el inicio de la primera página del i-nodo.

Una vez obtenido el mínimo, utilizaremos kmalloc() para obtener el puntero y copiaremos carácter a carácter desde el offset al puntero. Al igual que con write(), necesitaremos comprobar que hemos llegado al final de la página. Si se da esta situación, el offset pasará a ser la siguiente página del i-nodo. Finalmente, devolveremos el puntero devuelto por kmalloc().

changeDir(path)

Cambia la variable global *current_glob*, recordemos que referencia al directorio en el que se encuentra el usuario del SO, al directorio referenciado por path.

Tras obtener el i-nodo resolviendo el path, comprobaremos que el archivo es de tipo directorio. Tras la comprobación, simplemente asignaremos a *current_glob* el puntero a la primera página del i-nodo calculado.

2.11.6. Funciones de borrado

A la hora de borrar archivos necesitaremos realizar dos tareas. La primera, es limpiar cada una de las estructuras, de manera que no se pueda referenciar un archivo borrado. La segunda es liberar los recursos que fueron utilizados para su creación como las páginas utilizadas o el i-nodo reservado.

deleteFile(father, inFile, numChild)

Borra el archivo "inFile" del directorio "father". El argumento "numChild" es utilizado para evitar volver a buscar el archivo en el padre.

Lo primero que haremos será liberar el i-nodo. Para ello, actualizaremos la variable *minFreeInode* en caso de que el nodo liberado sea menor. Después, liberaremos las páginas que utilizaba y actualizaremos el i-nodo a libre. Finalmente, borraremos las referencias al archivo en el padre, vaciando la posición "numChild" del array *child* de "father". Con vaciando, nos referimos a igualar a 0 cada uno de los campos de *dir_entry*, desde el nombre del archivo hasta el número del i-nodo.

deleteDirContent(inFile)

Esta función eliminará el directorio "inFile" y realizando llamadas recursivas para borrar todo su contenido.

Recorreremos el array *child* de "inFile", el cual se encuentra en su página de memoria. Recordemos que la segunda posición de este array contiene la información del padre, por lo que deberemos evitar borrar su información.

Al recorrer el array, si la posición actual es un archivo regular, invocaremos a la función anterior. Sin embargo, si es un directorio, llamaremos a esta misma función con el hijo.

Finalmente, tras eliminar cada uno de sus hijos, liberaremos los recursos de "inFile", de la misma manera que hacemos en deleteFile().

delete(path)

Esta función eliminará el archivo referenciado por "path", independientemente de si es un archivo regular o un directorio utilizando las dos anteriores.

Comenzamos resolviendo el path para obtener el i-nodo. Tras esto, si el archivo es regular, llamaremos a deleteFile(), y si es un directorio, invocaremos a deleteDirContent(). En el caso de deleteDirContent(), borrará el contenido del directorio, pero no la referencia del i-nodo en su padre, por lo que deberemos vaciar la posición en el array *child* del padre.

Finalmente, si el archivo que hemos eliminado se encuentra en una posición intermedia del array, provocará que falle sucesivos borrados e impresiones. Para solucionarlo, copiaremos la información del archivo colocado en la última posición del array en el espacio generado por el archivo borrado.

2.11.7. Funciones de impresión

Estas funciones fueron creadas con propósito de depuración, y después utilizadas para crear comandos en la siguiente sección.

recPrintFs(inode, j)

Este método imprime recursivamente el contenido del sistema de ficheros a partir del directorio "inode". El parámetro "j" es utilizado para indicar el nivel de profundidad en el que nos encontramos, de manera que podamos indentar de manera correcta la impresión.

Obtendremos el i-nodo con get_inode(), y con el i-nodo conseguiremos el directorio de la misma manera que lo hacíamos en la función deleteDirContent(). Tras esto, recorreremos su array *child*. Es importante evitar los hijos "." y ".." para evitar ciclar indefinidamente. Finalmente, si el hijo de la posición del array es un archivo regular, imprimiremos su nombre, y si es un directorio, llamaremos recursivamente a esta función con el hijo y con "j" más uno.

printFs()

Imprime el sistema de ficheros al completo. Básicamente es una llamada a recPrintFs(), con el número de i-nodo 0 (recordemos que corresponde al nodo raíz) y el nivel 0.

listDirectory(path)

Imprime el contenido del directorio referenciado por "path".

Tras obtener el i-nodo resolviendo el path, comprobaremos que este sea un directorio. Si es así, simplemente queda recorrer su array *child* imprimiendo los nombres de los hijos, excluyendo "." y "..".

2.11.8. Funciones de los comandos

Los comandos de esta parte han sido creados como se describe en la sección 2.9.3. El método register_filesystem_commands() le asignará a cada comando su texto de ayuda, su clave y su

disparador. Por ejemplo, para mkdir, se realiza lo siguiente:

```
COMMAND mkdir;

void mkdir_function(int argc, char** argv){
    MARK_UNUSED(argc);
    createDir(argv[0]);
    return;
}

static void register_filesystem_commands(){
    mkdir.helpText = "Create a directory";
    mkdir.key = "mkdir";
    mkdir.trigger = mkdir_function;
    regcomm(&mkdir);
    ...
}
```

Por tanto, como se observa en el ejemplo anterior, cada disparador simplemente llamará a la función correspondiente del sistema de ficheros con los parámetros necesario. La correspondencia entre comando y método del sistema de ficheros es la siguiente:

- mkdir - createDir().
- mkfile - createFile().
- cd - changeDir().
- del - delete().
- lsall - printFs().
- ls - listDirectory().
- cat - read(): dado que cat imprime el archivo entero, el número de bytes que se utiliza como argumento es el resultado de getFileSize().
- echo - write().

2.11.9. Interfaz proporcionada para sobrescribir los métodos

Esta interfaz corresponde a la estructura *fs_interface*, y permite sobrescribir los siguientes métodos:

- write()
- read()
- delete()
- createFile()
- createDir()

- exists()
- printFs()
- getFileSize()
- changeDir()
- listDirectory()

Esta estructura se inicializa en fs_init(), asignando a cada método de la interfaz su método correspondiente de las secciones anteriores. Para utilizar esta interfaz, es obligatorio que los métodos creado devuelvan el mismo tipo y tengan los mismos tipos que los métodos originales.

Por tanto, para poder sobrescribir un método, hay que crear el método que lo sobrescribirá, y modificarlo la interfaz, la cual obtendremos con el método de get_FsInterface(). Un ejemplo de esto sería lo siguiente:

```
1
2  void myPrintFs(){
3      enrichedPrintLn("myPrintFs se ha ejecutado", &RED, NULL);
4  }
5
6  void kernel_main(uint32_t r0, uint32_t r1, uint32_t atags) {
7      [...]
8
9      fs_interface* aux = getFsInterface();
10     aux->printFs = myPrintFs;
11     aux->printFs();
12
13     [...]
14 }
```

El resultado de la ejecución de lo anterior de puede observar en la figura 2.22.

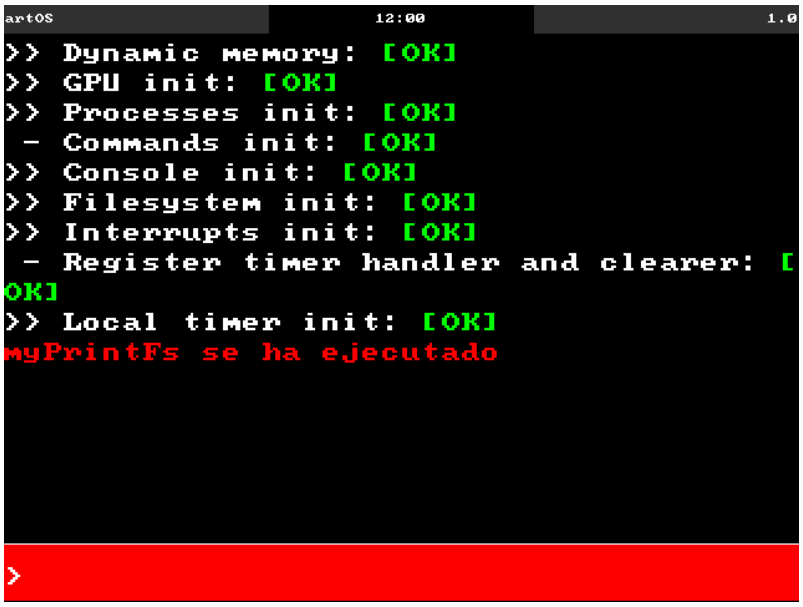


Figura 2.22: Ejemplo de uso de la interfaz del sistema de ficheros

Capítulo 3

Manual de uso

Dado que el desarrollo del proyecto se ha realizado en sistemas Linux, específicamente en **Ubuntu**, este manual se realizará enfocado a este último. Para poder ejecutar y probar el sistema operativo, necesitaremos:

- **La rama master del proyecto en Github**[1] que contiene un Makefile que construye el proyecto, el compilador necesario, un depurador, el código fuente y la documentación más relevante recolectada.
- **El emulador** es un software distribuido por QEMU, una serie de simuladores de procesadores, en concreto el de la Raspberry Pi 2. Se puede instalar utilizando el siguiente comando "*sudo apt install qemu*".
- **La Raspberry Pi 2** necesita una tarjeta microSD de memoria y el cable de alimentación correspondiente.

3.1. Reglas del Makefile

Situándonos con una terminal en la carpeta raíz del proyecto (aquella que contiene el Makefile), ejecutaremos el comando "*make <nombre de la regla>*", donde esta regla puede ser:

- **build:** Compila el proyecto, creando los object y la imagen del sistema operativo, y configurando lo necesario para ejecutarlo con el simulador. Tanto *run* como *debug* llaman a esta regla, por lo que no es necesario hacer build antes de ejecutar estas. La imagen del sistema operativo es *build/myos.elf*.
- **build_hard:** Realiza la misma compilación que *build*, pero adaptándola a la ejecución en la Raspberry Pi 2. La imagen pasará a llamarse *kernel7.img*. De esto se hablará en el capítulo 4.2.
- **variable_test:** Utilizado para imprimir todos los nombres de los archivos que componen el SO, mostrando los object, los assembly y los c.

[1] https://github.com/dacya/tfg1920-raspiOS

- **clean:** Elimina todo lo generado en el proceso de compilación, para eliminar archivos innecesarios cuando se termina de utilizar, como los object.
- **run:** Arranca el simulador con la imagen del sistema operativo, si existe. Si no, ejecuta *build* para generar la imagen. Es la regla que utilizaremos para probar el sistema operativo.
- **debug:** Actúa igual que *run*, excepto que se para en la primera línea de código. Para utilizarlo es necesario conectar el emulador con el depurador. Explicaremos cómo hacerlo y su uso en la sección 3.4 de este capítulo.

3.2. Ejecutar el Sistema Operativo

Emulador QEMU

Una vez compilado con *make build* (o simplemente *make*) como en la figura 3.1, se utilizará el comando *make run* desde la carpeta raíz del proyecto, lo que ejecutará el simulador y se abrirá la pantalla que simula la salida HDMI de la Raspberry Pi como en la figura 3.2.

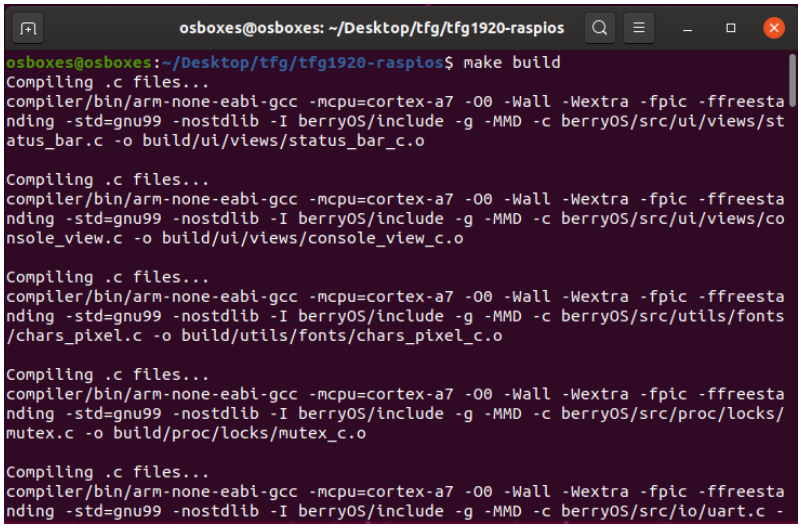


Figura 3.1: Ejecución de make build para compilar el SO para QEMU

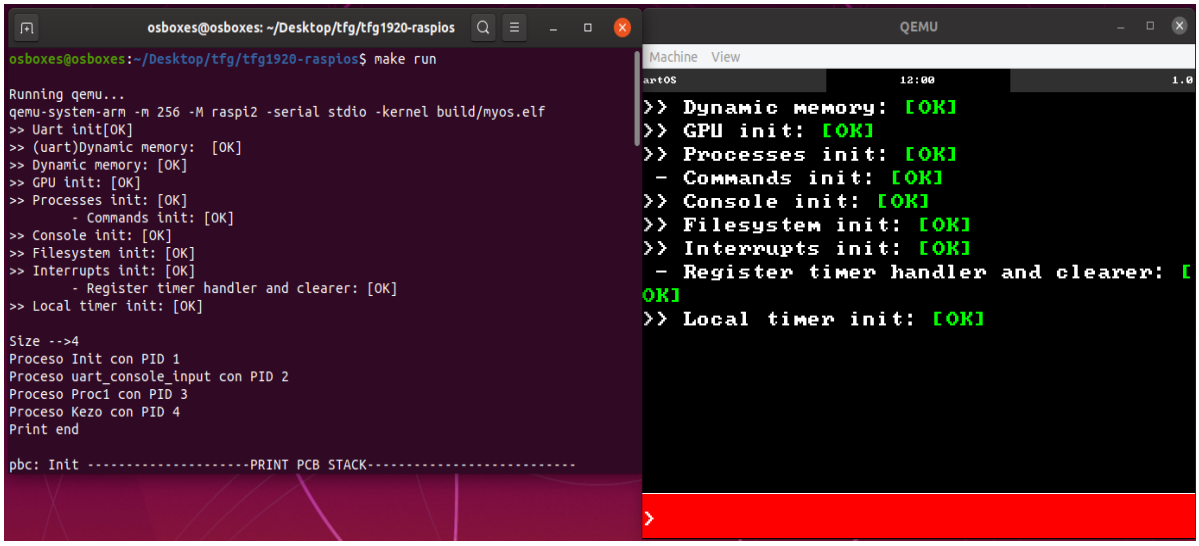


Figura 3.2: Ejecución de make run para lanzar SO con el QEMU

Para poder interactuar con ella, el método de entrada es a través de la UART, como se explica en la sección 2.4 del capítulo anterior. Por tanto, es necesario tener seleccionada la terminal donde ejecutamos el comando a la hora de escribir.

Finalmente, para ver las distintas cosas que podemos hacer con el SO, utilizaremos el comando *shcomm* para mostrar una lista de los comandos disponibles y su funciones.

Raspberry Pi 2

En este caso se debe usar el comando *make build_hard* que compilará el sistema operativo con el nombre *kernel7.img* en la carpeta *build*. Para ejecutar la imagen en la Raspberry es tan sencillo como borrar todos los archivos imagen y por último mover el nuestro a la tarjeta microSD.

La tarjeta microSD debe tener el sistema Raspbian [13] cargado en ella para reutilizar sus sistema de arranque y, así, facilitar el diseño y su puesta en marcha reemplazando la imagen del sistema.

3.3. Comandos disponibles

Los comandos han sido creados para mostrar las funcionalidades de algunos módulos de nuestro sistema operativo. Estos son:

1. **Memoria.**
   - **prheap:** imprime el espacio disponible en el heap, que como vimos en la sección 2.7, se utilizará para almacenar datos de pequeño tamaño, como un array de char. [Figura 3.3]

Figura 3.3: Salida del comando *prheap*.

- **prmem:** imprime un bitmap de las primeras sesenta páginas de la memoria. [Figura 3.4]

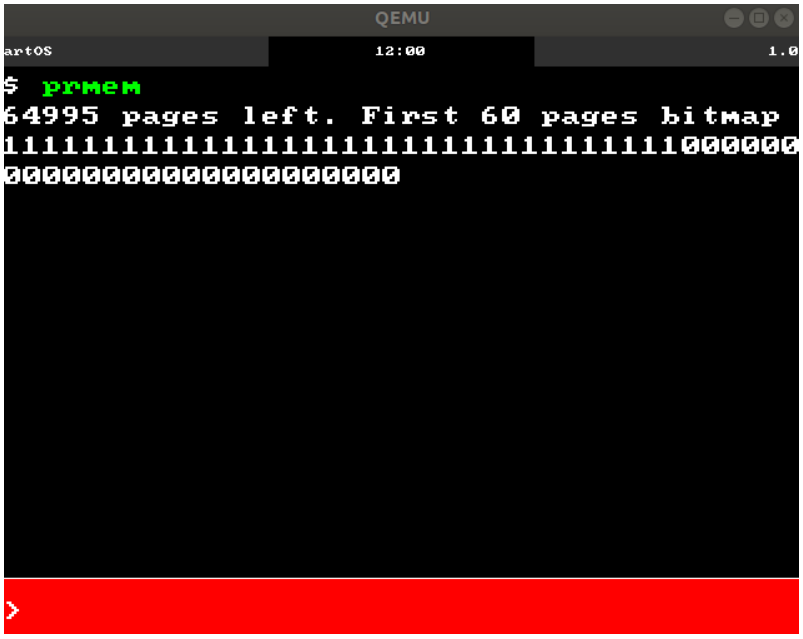
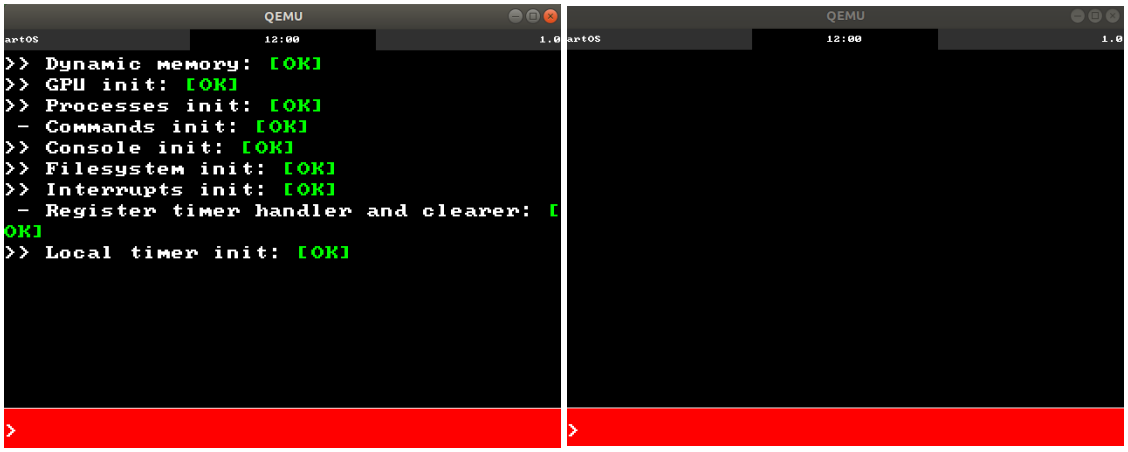


Figura 3.4: Salida del comando *prmem*.

2. **Interfaz gráfica.**

- **cls**: vacía la pantalla. [Figura 3.5]

Figura 3.5: Efecto de *cls*



(a) Antes de ejecutar *cls*

(b) Después de ejecutar *cls*

- **shcomm**: imprime los comandos disponibles y sus funcionalidades. [Figura 3.6]

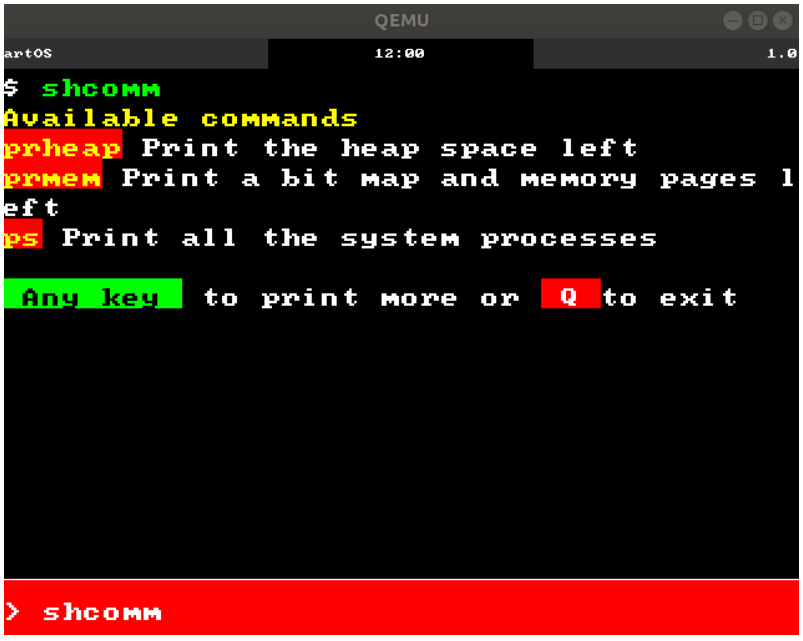


Figura 3.6: Salida del comando *shcomm*.

3. **Procesos:**

- **ps:** imprime el nombre y el PID de todos los procesos actuales. [Figura 3.7]

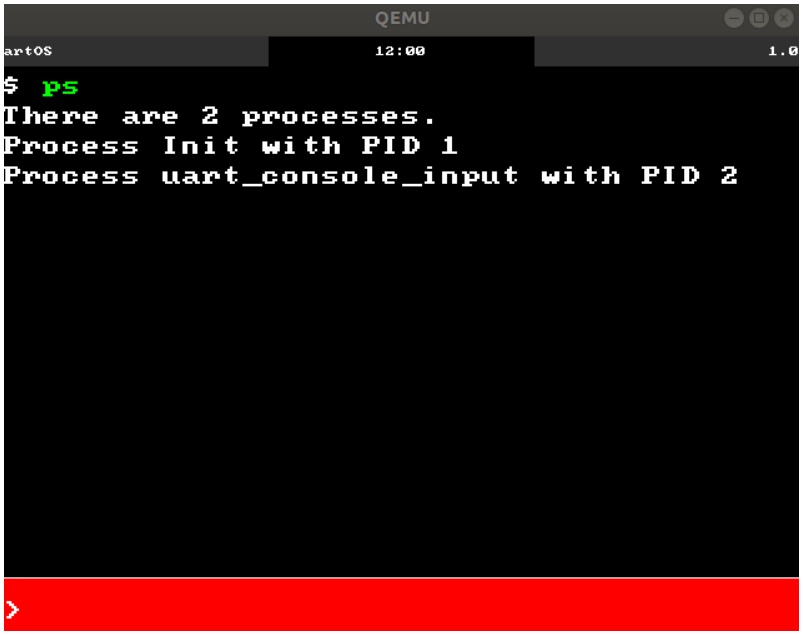


Figura 3.7: Salida del comando *ps*.

4. **Sistema de gestión de ficheros.**

   - **mkfile <ruta>:** crea en la ruta un archivo ordinario, sobre el que se pueden usar los comandos *cat*, *echo* y *delete*. [Figura 3.8]

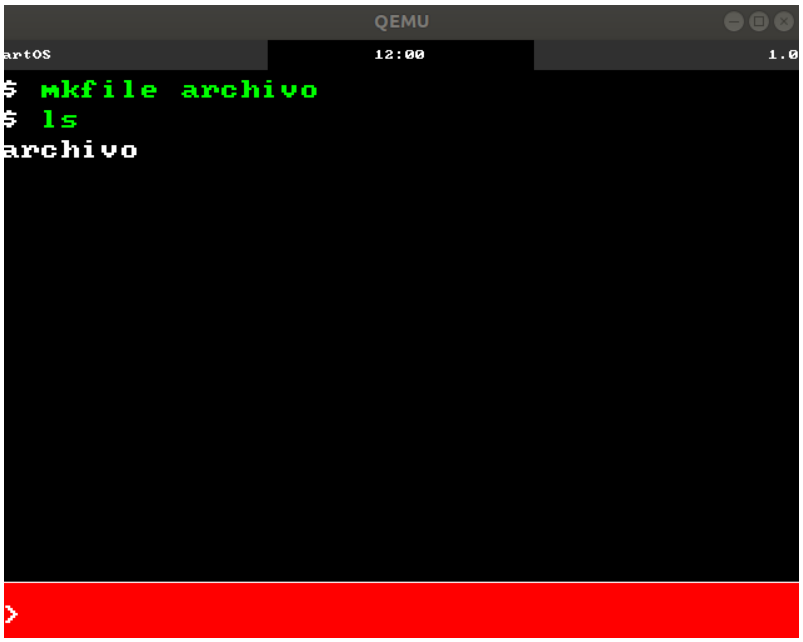


Figura 3.8: Efecto del comando *mkfile*.

   - **echo <texto><ruta>:** concatena un texto a un archivo ordinario. Este texto no admite espacios. [Figura 3.9]

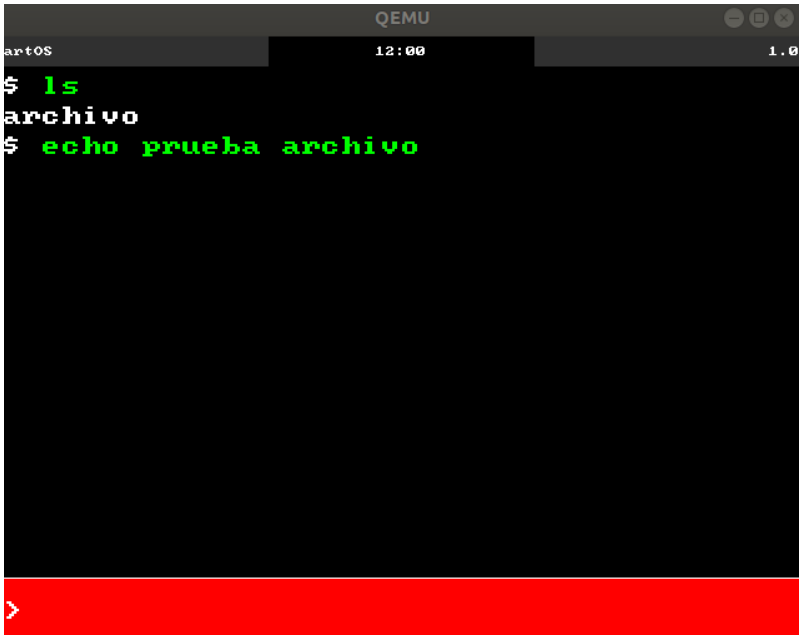


Figura 3.9: Ejemplo de uso del comando *echo*.

- **cat <ruta>:** imprime por pantalla el contenido de un archivo ordinario. [Figura 3.10]

Figura 3.10: Salida del comando *cat*.

- **mkdir <ruta>:** crea un directorio en la ruta, sobre el que se puede usar los comandos *cd*, *ls* y *delete*.[Figura 3.11]

Figura 3.11: Efecto del comando *mkdir*.

- **ls [<ruta>]:** lista los archivos contenidos en el directorio de la ruta. Si no se utiliza la ruta, se lista el directorio actual. [Figura 3.12]

Figura 3.12: Salida del comando *ls*.

- **lsall:** imprime toda la jerarquía de archivos desde *root*. [Figura 3.13]

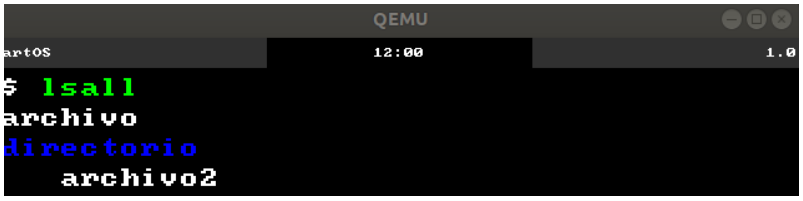


Figura 3.13: Salida del comando *lsall*.

- **cd <ruta>:** cambia el directorio actual al de la ruta. [Figura 3.14]

Figura 3.14: Efecto del comando *cd*.

- **del <ruta>:** borra el archivo de la ruta. Si el archivo es uno ordinario, simplemente lo borra. Si es un directorio, borra recursivamente su contenido. [Figura 3.15]

Figura 3.15: Efecto del comando *del*.

3.4. Utilizar el depurador

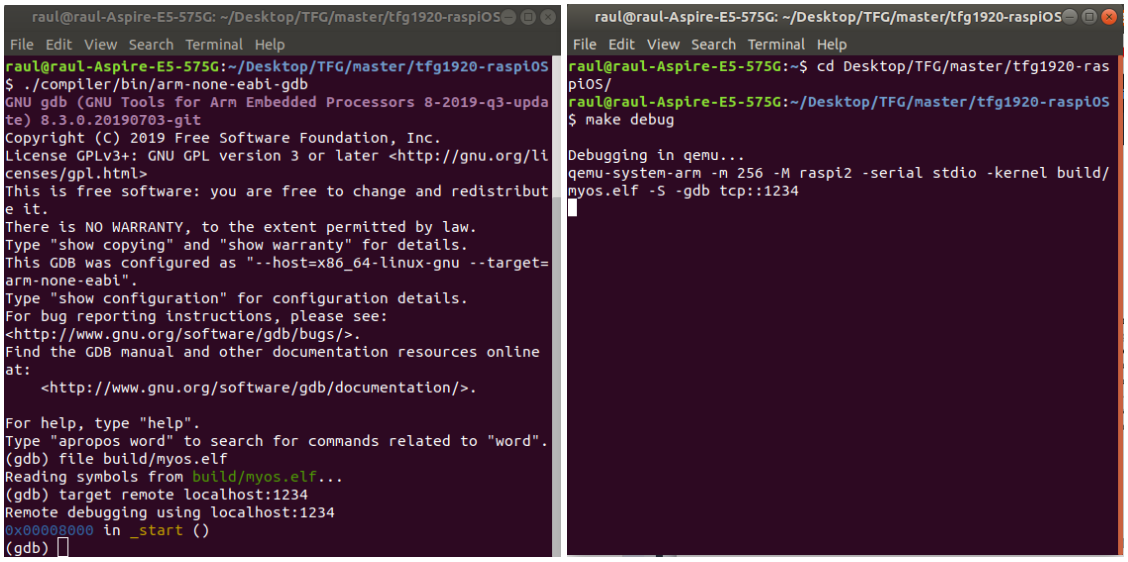
Para ejecutar el SO y ser capaces de emplear los beneficios de un depurador, necesitaremos dos terminales situadas en la carpeta raíz del proyecto. La primera se utilizará para utilizar el comando *make debug*, lo que abrirá un puerto para conectar un depurador. La segunda la utilizaremos para ejecutar GDB (GNU Debugger).

Para ejecutar GDB, utilizaremos el comando *./compiler/bin/arm-none-eabi-gdb*, lo que ejecutará el programa y nos mostrará una interfaz de comandos. Primero, necesitamos añadir los nombres de función, los nombres de variable, etc. Para ello, importaremos el ejecutable del SO con el comando *file build/myos.elf*. Tras esto, ya solo queda conectar el depurador a el emulador, para lo que utilizaremos en la misma terminal el comando *target remote localhost:<port>* siendo *<port>* el puerto en el que el emulador de QEMU espera para poder acoplarle un depurador[2]. Una vez finalizado todo lo anterior, ya podremos usar GDB como si depuráramos cualquier otro programa.

Un ejemplo exitoso del proceso anterior se puede observar en la figura 3.16. La subfigura 3.16a correspondería a la terminal en la que utilizaremos GDB, mientras que la subfigura 3.16b muestra la que contiene el proceso de QEMU.

[2]Esta información se muestra por pantalla al ejecutar el comando *make debug*

Figura 3.16: Ejemplo de inicio de depuración correcto.



(a) Terminal donde se utiliza GDB

(b) Terminal que contiene a QEMU

3.5. Repositorio git

El repositorio git está alojado en la plataforma *GitHub*: https://github.com/dacya/tfg1920-raspiOS

3.5.1. Estructura del repositorio git

Directorio raíz

El repositorio se compone de tres carpetas:

- La carpeta del código del sistema operativo.
- Los archivos del compilador y su documentación.
- La documentación oficial distribuida por el fabricante que hemos ido recogiendo del proyecto.

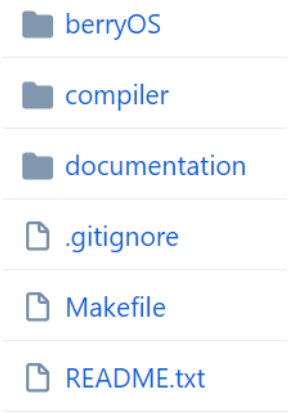


Figura 3.17: Directorio artOS

Directorio del código fuente

Con respecto a la carpeta con todo el código fuente se estructura de la siguiente forma:

- **src** contiene los módulos en código C.
- **include** recoge los archivos cabecera *.h* de todos los módulos *públicos*.
- **arch** aloja las dependencias de arquitectura en código ARM.
- **LICENSES** contiene las licencias del código.

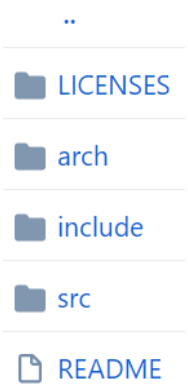


Figura 3.18: Directorio artOS

3.5.2. Aportaciones al repositorio

Si se trabaja dentro del repositorio se debe seguir la técnica *git-flow* donde el desarrollador trabaja en una rama clonada de *master* o *wip* y posteriormente debe abrir una petición de fusión (*merge request* o *pull request*) a la rama de desarrollo *wip* para que sea analizada y fusionada correctamente. Para cada versión finalizada en *wip*, previamente testeada, se fusiona con *master* pasando a ser la última versión estable.

Para cambios de repositorios externos o *forks* se debe pedir una petición de fusión (*merge request* o *pull request*) para que sea analizado por los desarrolladores y posteriormente aceptado.

IMPORTANTE: Todas las funciones expuestas por un módulo deben ir documentadas siguiendo el formato JavaDoc.

Capítulo 4

Conclusiones y trabajo futuro

4.1. Conclusiones

En este Trabajo Final de Grado hemos desarrollado un Sistema Operativo completamente funcional para la Raspberry Pi 2. Para facilitar el diseño y su puesta en marcha, se ha reutilizado el complejo sistema de arranque de Raspbian 4.19. **Los módulos desarrollados** se describen a continuación:

El kernel es el módulo principal de nuestro sistema operativo, es el encargado de inicializar correctamente y en el orden adecuado el resto de módulos y configuraciones del hardware. Este modulo se ejecuta tras el proceso de arranque.

El módulo de gestión de E/S fue uno de los primeros módulos en definirse y permaneció activo durante todo el desarrollo del proyecto. Con este módulo somos capaces de recibir y enviar datos a través de la UART. Además, hemos creado una librería que facilita el uso del banco de pines de entrada y salida característico de la Raspberry Pi. Gran parte del desarrollo de este TFG depende de la depuración, sin este módulo no habríamos sido capaces de crear este sistema operativo.

El módulo de gestión de interrupciones y excepciones nos permite tratar cualquier tipo de evento que interfiera en la tarea del CPU. El objetivo de este módulo consiste en dar soporte para las interrupciones, pero durante su desarrollo nos dimos cuenta de que las excepciones se tratan igual, por lo que se ha dejado la base para tratarlas. A través de este módulo gestionamos uno de los relojes hardware para que interrumpa periódicamente la ejecución de cualquier proceso, funcionalidad esencial para el módulo de gestión de procesos.

El módulo de gestión de memoria proporciona la organización de toda la memoria accesible por el CPU y la capacidad de reservar espacios de memoria sin conflictos entre los distintos módulos. Hemos tenido en cuenta el método que tiene el arranque de la Raspberry Pi 2 para obtener la cantidad de memoria disponible a través de los ATAGS.

El módulo de gestión de procesos consiste en el seguimiento de distintos programas a los que

asociamos una PCB (de inglés Process Control Block). Como se ha mencionado, con el módulo de gestión de excepciones e interrupciones se ha diseñado un planificador que permite la concurrencia entre todos los procesos dentro del sistema operativo con una interfaz para desarrollar fácilmente nuevas estrategias.

El módulo de gestión de sistema de ficheros crea una capa de abstracción para el manejo de información dentro del sistema operativo en forma de ficheros y carpetas. Este módulo está basado en la representación del nodo indexado que implementa GNU/Linux.

El módulo de la interfaz gráfica aprovecha el canal de mensajería mailbox de la arquitectura para comunicar la tarjeta gráfica con el procesador. Se obtiene así una sección compartida de memoria por la cual la CPU puede gestionar a nivel de píxel la información transmitida por el puerto HDMI formando así la librería GPU. A partir de ésta hemos desarrollado un sistema de vistas similar al de Android con el que se implementa una consola con comandos que trabaja en un proceso propio.

Finalmente, el módulo de sincronización ofrece cerrojos para ofrecer atomicidad en secciones críticas, necesario para el procesamiento concurrente.

A lo largo del desarrollo de este TFG hemos aplicado los conocimientos que hemos adquirido en asignaturas como: Sistemas Operativos, Ampliación de Sistemas Operativos, Estructura de Datos y Algoritmos, Ingeniería del Software, Estructura de Computadores, Desarrollo de Sistemas Interactivos, etc. Además de la parte más técnica, también se ha aprendido a trabajar en equipo y se ha explotado al máximo el trabajo en paralelo con herramientas de control de versiones y con especial atención al diseño ya que la división en módulos es esencial para el trabajo en paralelo sin conflictos.

También hemos adquirido nuevas habilidades como la capacidad para filtrar información útil en la documentación del fabricante, profundización del funcionamiento de arquitecturas ARM así cómo en su programación y la capacidad de adaptación a las circunstancias.

4.2. Trabajo futuro

Creemos que el proyecto que hemos llevado a cabo cumple con las especificaciones mencionadas en la sección 1 de introducción y a partir de estas bases nos gustaría dar posibles ampliaciones o posibles proyectos como **trabajo futuro**.

En primer lugar, se debería dar soporte para distintos modos de ejecución, como por ejemplo, soporte para memoria virtual o seguridad.

En segundo lugar, se han sentado las bases para crear una interfaz sencilla y eficiente que facilite el registro de rutinas en C para el tratamiento de interrupciones. Se deberían explotar estas bases.

En tercer lugar, el sistema de gestión de ficheros se puede extender en sus capacidades para poder dar soporte a enlaces rígidos y enlaces simbólicos, permisos para leer, ejecutar o escribir en un fichero, etc.

En cuarto, lugar el módulo de procesos se podría ampliar creando distintos planificadores gracias a que se ha implementado una interfaz de abstracción para ello.

En quinto lugar, con respecto a los módulos relacionados con la interfaz gráfica, se podría dar soporte para la realización de cálculos directamente en la tarjeta gráfica liberando de carga a la CPU. Además, se ha dejado el sistema de vistas preparado para crear un sistema de ventanas.

Finalmente, el hardware de la Raspberry Pi ofrece muchas más posibilidades de las que hemos aprovechado en este TFG, nos gustaría remarcar que es un buen trabajo futuro permitir la implementación para el manejo de puertos USB, Ethernet, sistema de audio, Bluetooth y WiFi entre muchas otras cosas. También el SO podría usarse de herramienta en otras ramas de trabajo como por ejemplo en la robótica.

Capítulo 5

Contribuciones al proyecto

Alejandro Cancelo Correia

Desde el principio nos dimos cuenta de que el diseño de funcionalidades base dependían mucho del lenguaje ensamblador por lo que mi tarea a lo largo de este TFG ha sido dedicarme a comprender la arquitectura lo máximo que pudiese en el tiempo que tuvimos y aplicar esos conocimientos en los distintos módulos que lo requiriesen. Para ello utilicé los documentos que podéis encontrar en el fichero berryOS/documentation/ y largas horas vagando por foros y webs especializadas en diseños bare metal en busca de más conocimiento.

Una de mis primeras tareas fue comprender e implementar el proceso boot, aquí he de aclarar una cosa, la subrutina en ensamblador del boot la obtuvimos del tutorial que nos ofreció nuestro Director de TFG José L. Risco Martín, mi participación en esa subrutina fue para hacerla más descriptiva y para permitir la inicialización de un componente base de mi siguiente tarea, las interrupciones, pero antes me gustaría mencionar que durante el desarrollo del boot y de la subrutina kernel_main (la cual se encarga de inicializar los demás módulos) se me ocurrió aplicar un estándar de inicializaciones el cual seguimos a lo largo del TFG.

Mi siguiente tarea fue crear el módulo de interrupciones. Esta tarea, aunque sencilla, requería que aprendiese acerca de todos los tipos de excepciones que la arquitectura del Cortex-A7 podía ofrecer, por lo que no solo me dediqué a las interrupciones, también me dedique a preparar todo tipo de excepciones para que alumnos futuros solo tengan que editar los cambios para que se adapten a los nuevos módulos que vayan a implementar. También desarrollé una librería con las interrupciones más importantes de las 72 posibles existentes en el CPU y definir una rutina de tratamiento en C que se ejecutase por cada interrupción.

A continuación pasé a desarrollar un reloj que disparase interrupciones periódicamente, aprovechando los distintos componentes hardware que ofrecía el Cortex-A7. Nuestra primera idea era usar la propia Raspberry Pi para ello, pero debido al gran trabajo que nos suponía y que queda fuera del alcance de este proyecto, decidimos que la mejor idea era enfocarnos en el emulador de QEMU, por lo que tenía que investigar el soporte que daba QEMU para los relojes dl CPU, aquí me ayudó nuestro Director de TFG José L. Risco Martín, ya que era consciente de que QEMU únicamente daba soporte a lo que se conoce como *System Timer*, unos relojes que pertenecían a cada núcleo y del cual hablaré más tarde.

Una vez acabada la implementación del reloj y ver que todo funcionaba a la perfección, decidí incluir una pequeña librería en C, con la que abstraer y facilitar el uso de dicho módulo.

En este punto del desarrollo, relevé a mi compañero Raúl Sánchez Montaño de sus responsabilidades con el módulo de procesos del kernel y me di cuenta de varias cosas. Mi compañero basó sus diseños en el tutorial que ya he mencionado previamente, el problema era que el que creó el tutorial no comprendía bien qué era un proceso y qué responsabilidad tenía el kernel, sus diseños consistían en entender a los procesos como subrutinas, en vez de como elementos separados sin ningún tipo de dependencia, bajo esa premisa la subrutina de kernel_main se entendía como un proceso, cosa que no puede estar más lejos de la realidad, por lo tuve que deshacerme de algunos de sus diseños como: el diseño de la estructura del PCB, los cambios de contexto y el planificador e implementarlo de cero dando soporte a la verdadera asincronicidad que tienen los procesos. Además implementé una pequeña interfaz para registrar planificadores.

Una vez resuelto este problema, nos surge la situación de que ahora sí tenemos asincronicidad, por lo que también podríamos tener condiciones de carrera así que mi última tarea consitió en crear cerrojos que permitiesen proteger las secciones críticas del código.

En resumen, mi contribución a este TFG ha sido el contacto directo con la arquitectura del CPU creando funcionalidades en ensamblador, apoyando a esas funcionalidades con algún código en C o librería en C diseñado por mi y entender el flujo de ejecución de la propia arquitectura.

Tomás Golomb Durán

Mi trabajo en el proyecto respecto a la implementación se ha centrado en el bloque de módulos de entrada y salida que culmina en una liberaría estándar (*stdio.h*), en un sistema de vistas que compone la interfaz de usuario y varios sistemas de comunicación a través del banco de puertos de la placa.

Empecé con la librería de control GPIO que tiene como objetivo abstraer el uso del banco de pines que incluye la placa para aplicaciones de comunicación entre dispositivos y depuración en la placa, pero, también, pensando en el futuro, para que otros alumnos puedan utilizarla como herramienta para el desarrollo de aplicaciones robóticas. A raíz de este último punto, se me ocurrió que las funciones expuestas por el módulo deberían estar bien documentadas, por eso impulsé el uso del formato de documentación JavaDoc en todo el proyecto.

Posteriormente, revisé el módulo que mis compañeros desarrollaron sobre la UART reutilizando la librería GPIO ya que este dispositivo de comunicación de carácter general transmite los datos a través del banco de pines.

Mi siguiente objetivo fue buscar el uso de la GPU para poder desarrollar una interfaz de usuario propia del sistema a través del puerto HDMI, ya que, hasta el momento, la salida de datos a través de la UART mantiene la dependencia con otro sistema operativo de una consola bash. Para ello, tuve que buscar información a través de foros, *wikis* y el repositorio de *Linux* porque la documentación del fabricante estaba incompleta. Finalmente, conseguí implementar el protocolo a través del sistema de intercambio de mensajes mailbox entre la CPU y la GPU para la creación de un framebuffer, una estructura de datos compartida entre los dos componentes para renderizar

imágenes. Esto funciona correctamente en el emulador QEMU, pero no se han conseguido lograr los mismos resultados en el hardware que necesitan largas horas de depuración que se vieron interrumpidas a causa de la pandemia del COVID-19.

Una vez conseguido el framebuffer, tenemos acceso a la lectura y escritura de cada píxel de una pantalla conectada al puerto HDMI. Consecuentemente, implemento un sistema de vistas basado en el diseño del sistema operativo de código abierto Android. Se convirtió en un reto bastante complejo porque el lenguaje de programación C no es orientado a objetos y por tanto no existen las técnicas de herencia, polimorfismo y clases abstractas como si aprovecha Android con java, por lo que tuve que ingeniar estas técnicas. De esta forma, surge un módulo de interfaces de usuario (IU) con el que soy capaz de implementar una consola con un componente de entrada de datos y una barra de estado.

En este momento pienso que es necesario el diseño de una librería estándar entrada y salida, conocida popularmente como stdio.h, que actualmente redirige la salida de datos a través de la UART y la consola gráfica. Esta abstracción (o fachada) facilitaría a futuros desarrolladores redireccionar el flujo de datos sin la obligación de *refactorizar* el proyecto.

Una vez terminada la consola, me propongo el desarrollo de un sistema de comandos con paso de parámetros lo suficientemente abstracto para que cada módulo pueda adaptarse y crear el suyo propio. Este módulo tiene especial interés técnico porque le afecta el orden en como se inicializan los módulos por el kernel pudiendo así afectar al sistema entero. Esto se debe a que se pueden encontrar dependencias entre módulos en ambos sentidos, por lo que uno se encontrará que el otro no está inicializado.

En el apartado de miscelánea, he implementado una lista dinámica doblemente anidada con un nodo fantasma que incluye varias funcionalidades intentando que sea lo más eficiente posible y usando directivas del compilador para conseguir genericidad. Ésta se ha utilizado prácticamente en todos los módulos: memoria, procesos, vistas, comandos, etc.

A raíz de varios problemas que se nos presentaron al trabajar en paralelo, mis compañeros me asignaron el rol de director de proyecto por tener más experiencia con el uso de sistemas de control de versiones y, en concreto, git y GitHub. Me encargaba de repartir y gestionar las tres líneas de trabajo y el análisis de las peticiones de fusión de código para solucionar los conflictos al fusionar las ramas del repositorio git.

En relación a la redacción de la memoria, he redactado de la parte del diseño los módulos ya mencionados, el capítulo introductorio, el resumen y palabras clave y he ayudado con la revisión de varias secciones del proceso iterativo.

En resumen y para concluir, he desarrollado el apartado de entrada y salida de datos pasando por implementaciones a bajo nivel (la UART, la GPIO y la GPU) y “escalar” un poco más con el desarrollo de un sistema gráfico de vistas basado en Android que se comunica a través del puerto HDMI para romper la dependencia de tener que trabajar con otro sistema operativo. A raíz de estos módulos he programado una consola con un sistema de comandos con paso de parámetros

adaptable a cualquier módulo y he facilitado una librería estándar de entrada y salida para poder redireccionar el flujo de datos. Además de codificar, he adquirido un rol de gestión de trabajo y repositorio para evitar los conflictos de trabajar en paralelo.

Raúl Sánchez Montaño

Tras una de las primeras reuniones, y tras recibir de parte de nuestro director de TFG, José L. Risco Martín, la tarjeta de memoria y la Raspberry Pi 2, decidimos que me encargaría de las pruebas en hardware. Debido a esto, una de mis primeras tareas consistió en conseguir el firmware necesario en la tarjeta de memoria para ejecutar nuestro SO en la Raspberry Pi. Además, también investigué, junto a mis compañeros, como funcionaba las distintas partes del proceso de arranque de esta placa.

Tras esto, y basándome en el tutorial[1] de Jake Sandler aportado por José L. Risco Martín, hice una primera aproximación al SO la cual ejecutaba con éxito la fase de boot, convirtiendo la Raspberry en un sistema monoprocesador, y conseguía hacer parpadear un LED utilizando la UART.

Tras esto, y continuando con el tutorial, realicé los módulos de memoria y memoria dinámica. Para realizar la parte de memoria, tuve que informarme de cómo se pasaban datos sobre el hardware al software en la Raspberry Pi, ya que necesitaba saber cuánta memoria tenía disponible. Esto se hace mediante los atags, así que creé un archivo que contenía todas las estructuras relacionadas con los atags, aunque finalmente solo implementé el método que devolvía la cantidad de memoria disponible.

Volviendo al módulo de memoria, y fijándome en el tutorial, cree el método de inicialización que dividía la memoria en páginas, y dividía estas en tres secciones, además de corregir varios errores de concepto del tutorial. En este proceso, también inicialicé los metadatos necesarios para el correcto manejo de las páginas. Además, implementé métodos que permitían obtener páginas de memoria para ser utilizadas en otros módulos, y liberarlas una vez finalizado su cometido.

Tras el módulo anterior, en el mismo archivo, comencé con el módulo de memoria dinámica. Para ello, inicialicé una de las secciones del módulo anterior para obtener una zona de memoria para extraer fragmentos del tamaño requerido. Además, implementé métodos basados en el tutorial para obtener fragmentos y liberarlos, modificados para solucionar problemas con el tratamiento de punteros. Estos métodos también realizan un proceso de desfragmentación de la memoria.

Mientras trabajaba en los módulos anteriores, realizaba pruebas en hardware para probar distintos módulos que había creado mi compañero Tomás Golomb Durán. Estos módulos son la GPIO, encargada de manejar los pines de la Raspberry Pi, y el módulo encargado de manejar el HDMI. Además, hice pruebas para intentar comunicarme con la Raspberry Pi a través de la UART física, utilizando un cable USB-TTL. Pese a que tanto la UART como el HDMI funcionaban perfectamente en el simulador, todas las pruebas en hardware fueron infructuosas, por lo que unánimemente, y tras consultar a nuestro director, decidimos que a causa de la pandemia y las dificultades de no poder depurar en grupo nos centrásemos en implementar para el simulador.

[1] https://jsandler18.github.io/

El siguiente módulo en el que participé fue el de procesos. Basándome en el tutorial nuevamente, implementé las estructuras que almacenarían los datos de la PCB, además de los métodos para inicializar las estructuras y elementos necesarios. También implementé los métodos para crear un nuevo proceso y borrar un proceso acabado, además de una función de impresión para depurar el código. En relación al cambio entre procesos, utilicé y manejé el código ensamblador del tutorial para obtener una primera aproximación. Tras ejecutar el código y depurar, comprendimos que había un gran fallo de concepto en el diseño del cambio de procesos. A partir de este momento, tomó el relevo mi compañero Alejandro Cancelo Correia, ya que tenía más conocimientos del lenguaje ensamblador, dado que había implementado el módulo de interrupciones, el cual contiene una gran cantidad de ensamblador. Debido a la importancia de este módulo, y a las complejidades generadas por él, todo el grupo participó en su depuración varias veces, ya que encontramos errores y fallos de concepto en varias ocasiones.

El último módulo que implementé fue el sistema de gestión de archivos. Para realizarlo, busqué información de las distintas estructuras que utiliza un sistema basado en i-nodos, como ext2, así como la manera en la que gestiona la estructura de directorios. Tras investigar lo suficiente implementé las estructuras necesarias y la tabla de i-nodos, así como la inicialización del directorio raíz y los metadatos de los i-nodos. Tras esto, implementé los métodos básicos, crear y borrar directorios y archivos y leer y escribir archivos. Además, para darle más funcionalidades, añadí métodos para cambiar de directorio y listar archivos, además de crear varias funciones de impresión para depurar el proceso.

Después de realizar lo anterior, y para equiparar un poco más el sistema a un sistema de gestión de archivos común, añadí los procesamientos necesarios para permitir que hubiesen paths relativos y absolutos. Finalmente, y para permitir que el código fuese más usable y editable, añadí una interfaz en la que el usuario puede implementar sus propio métodos, sobrescribiendo lo creados por defecto.

La última parte que implementé fueron los comandos que se ejecutan desde la consola. Tras la realización de la consola y del mecanismo para crear y registrar comandos por parte de mi compañero Tomás Golomb Durán, se permitía implementar comandos propios. Por ello, decidí implementar distintos comandos en cada uno de los módulos en los que había participado, de manera que el usuario final del SO pudiese realizar acciones como manejar archivos o ver una lista de los procesos en ejecución. Para crear estos comandos, la mayoría de las funciones ya las había implementado, aunque también tuve que crear otras y adaptar algunas.

En resumen, mi aportación al TFG fueron: las pruebas en hardware iniciales, los módulos de memoria y memoria dinámica, el módulo de gestión de archivos y la implementación de varios comandos para la consola.

Y para terminar, también me gustaría comentar que ha habido un tiempo dedicado al aprendizaje del manejo de Github y el protocolo git. Este aprendizaje no es solo a nivel individual, si no también a nivel cooperativo, ya que manejar un proyecto tan amplio, y además, tan modularizado, ha provocado que le tuviésemos que sacar el máximo partido posible a la gestión de ramas.

Apéndice A

Introduction

This Bachelor's Degree Final Project is focused on the design and implementation of an Operating System for the Raspberry Pi 2, the second version of the well-known series of single-board computers made by the english Raspberry Pi foundation [7].

This project has consisted in the design of the essential modules that define an operating system. These modules can be grouped into three sections:

- The **physical memory management** is the essential section that support most of the modules. Its components are in charge for the distribution of the available memory between the rest of the modules without confrontations. For example, the section's modules take care of the processesñeeds, allowing concurrent access to data without race conditions.
- The **processing control** section is in charge of rationing the processor executing power using the processes management system in combination with the interrupt handling system, allowing to choose between different schemes for adapting to the eventual needs.
- The **input/output** module offers many alternatives for data reception and transmission, for example, the GPIO pins or the graphical user interface available through the HDMI port and so breaking the dependency with an auxiliary OS.

A.1. Objectives

The project's main goal is to develop an open source with educational purposes Operating System for the Raspberry Pi 2 called artOS.

This project can be used by the community as a basics tool in the Operating Systems based subjects with the main goal of simplify and speed up the process of renewing and improving the curriculum and practical activities of these. The project is structured to allow modifications in an easy way, giving priority to the simplicity and understanding. Also, it's important that the OS could be launched in an emulator, so we can easily make changes, tests and debug errors and that's why we chose QEMU.

The developed operating system achieves the following requisites:

- Basic support for the ARM architecture.

- Minimal input/output system, with a interrupt management layer and a basic graphical user interface.
- Multi-threading execution with locks that allows concurrency.
- Memory management and a file system module with basic features.

A.2. State of the art

The Raspberry Pi foundation has the objective of encouraging the education of computer science. Without a doubt, it has been reached considering this project existence, and the fact that this device has transcended beyond, becoming a general purpose microprocessor that stands out due to its extensive use in robotics, due to the bank of pins that all previous and future versions have in a lateral of the board. In addition, it has been developed a great community around forums, blogs, and websites which has result in several open source tools.

The first Raspberry Pi was released in February 2012. After its release, in June that same year, was released the first totally compatible OS with Raspberry PI infrastructure, whose name is Raspbian [15]. Besides, the interest in operating systems designed for Raspberry Pi has been increasing thanks to the rise of IoT, so that even Microsoft released its own compatible OS [10] with Raspberry Pi.

As regards the educational environment, there are some attempts of open source development with academic purpose, like our project. An example is this Cambridge University project [11] which tries to implement the OS in assembly code for the Raspberry Pi first version. Nevertheless, our project is programmed in C and assembly language, which makes it more legible. Besides, we have progressed further than the scope of this attempt.

Another related project, which we have used as a guide as it has been designed for the Raspberry Pi 2, developed by Jake Sandler [17] from Google. He explains in a very free-flowing way the procedures, although he made some mistakes. These range from simple errors like a bad management of pointers to a big misunderstanding related with processes. Besides, we have developed further than this attempt vision, implementing a graphical user interface with commands and a file system.

Lastly, the two previous attempts don't make an effort to improve the code's usability, while we have created libraries and interfaces in order to ease future development.

A.3. Methodology and work plan

Along with our project director, José L. Risco Martín, we decided to conduct periodical meetings in which the progress was verified and the following tasks considered. As a result of COVID-19 pandemic and the state of alarm decree, we kept the meetings but telematically, using the *Meet* application made by Google. In relation to the communication between members, it has been used messaging apps and, before the pandemic, we used to gather in the Computer Engineering Faculty's library group rooms and laboratories.

As the group is formed by three components, the work has been distributed in a development branch for each student. To achieve this, *git* has been used for versions control and a *GitHub* repository to save and manage our progress. Besides, to ease the use of the modules between members, the code has been commented using the same syntax defined by *JavaDoc*.

Regarding this document, each member has explained technically his work in the Design chapter 2. The rest of the chapters have been equally distributed for it's development with some iteration between the students and the director José L. Risco Martín.

A.4. Document structure

This document collects the development process of the operating system, how to compile and execute it and the final conclusions. The structure is as it follows:

1. The second chapter explains technically the design with implementation details and usage examples of the Operating System modules.

2. The third chapter shows how to compile and run the operating system and the included features in it.

3. In the fourth chapter explains the final conclusions and the future lines of work are indicated.

4. Lastly, the fifth contains the summary of each member contributions.

Apéndice B

Conclusions and future work

In this Bachelor's Degree Final Project we have developed an utterly functional Operating System for Raspberry Pi 2. To ease the design and to start as soon as possible with the project, we have reused the Raspbian 4.19 boot process. The developed modules are explained below:

The kernel is the main module of our Operating System, it is in charge of initializing correctly and in order the rest of the modules and hardware configurations. For this reason, the kernel is executed after the boot process.

The I/O management module was one of the first implemented sections along with this project. With this module we can send and receive data through the UART. Besides, we have created a library that makes easier the usage of the input/output bank of pins characteristic of Raspberry Pi. A huge part of this project development depends on debugging, so, without this module, we wouldn't have been able to create this operating system.

The interrupts and exceptions module allows us to process any event that interfere with the CPU's normal execution. The goal of this module is to support interrupts, but as the rest of exceptions are processed in the same way, we also have created what is necessary to support all type of exceptions. With this module we are able to use a physical clock to perform periodically interruptions to any of the processes executing, which is essential for the processes management module.

The memory management module provides organization of the memory that can be accessible by the CPU, and the capability to allocate memory without conflicts among modules. We use Atags to retrieve the available memory, a Raspberry Pi protocol to pass hardware system data when booting up the system.

The processes management module keeps the track of programs that have a PCB (Process Control Block) associated with them. As mentioned before, using the interrupts and exceptions module we have designed a scheduler that allows concurrency between processes.

The file system management module creates an abstract layer to manage data inside the operating system with files and directories based on the indexed node file system implemented in GNU/Linux.

The graphical user interface module takes profit of the mailbox channel from the architecture to communicate the graphics card with the processor, obtaining a shared section that can be used by the CPU to send information at pixel level to be displayed through the HDMI port. These operations are the base of the GPU library that uses the graphical user interface module. By using this library, we have developed a system based on the Android views system, which results in a command console running on its own process.

Lastly, the synchronization module allows atomicity on critical sections using locks, essential for concurrent processing.

In this project, we have used the knowledge learnt in subjects such as: Operating Systems, Advanced Operating Systems and Networks, Data Structures, Fundamentals of Algorithms, Software Engineering, Computer Organization, Interactive Systems Development, etc. Besides the technical part of the project, we have also applied teamwork and parallel work using version control systems and a modular design that prevents conflicts during the implementation.

We also have acquired new abilities like gathering and filtering useful information through the official manual documentations, a better knowledge of the ARM architecture and its programming and look for solutions in unexpected situations.

B.1. Future work

We think that the project achieve the specifications of introduction's 1 section, and using these as basis we want to propose some future projects:

First of all, should be extended the support for different execution modes, as virtual memory or security.

In second, use the implemented interrupts to create a simple, efficient interface that make easier to register C methods for interruption handling.

In third place, improve the file system module to support hard links and symbolic links, file permissions, etc.

In fourth place, implement different schedulers using the processes module interface.

In fifth place, related to the graphic user interface module, give executing support to the GPU in order to reduce the CPU load. Furthermore, it can be implemented a windows system using the views and view groups framework.

Lastly, the Raspberry Pi hardware offers a lot of possibilities that can be used, as the USB ports, Ethernet, the sound system, Bluetooth, WiFi and others. Also the OS can be used as a tool in other branches like robotics.

Bibliografía

[1] Android. *Código fuente de ViewGroup.java*. https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/java/android/view/ViewGroup.java.

[2] Android. *Código fuente de View.java*. https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/java/android/view/View.java.

[3] Android. *Layouts system overview*. https://developer.android.com/guide/topics/ui/declaring-layout.

[4] ARM. *Official developer ARM documentation*. https://developer.arm.com/.

[5] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Really, 2000.

[6] Raspberry Pi foundation. *Proceso para utilizar sistemas operativos en la Raspberry Pi*. https://www.raspberrypi.org/documentation/installation/installing-images/.

[7] Raspberry Pi foundation. *Página principal de Raspberry Pi*. https://www.raspberrypi.org/.

[8] GNU. *Uso de la regla make de GNU*. https://www.gnu.org/software/make/.

[9] Christopher Hinds and William Hohl. *ARM Assembly Language: Fundamentals and Techniques*. CRC Press, 2nd edition, 2014.

[10] Microsoft. *Sección de IoT de Microsoft*. https://developer.microsoft.com/es-es/windows/iot/.

[11] Department of Computer Science and Technology of Cambridge. *Baking Pi, curso para el desarrollo de un Sistema Operativo para Raspberry Pi*. https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/.

[12] Raspberry Pi. *BCM2837 ARM Peripherals*. https://github.com/raspberrypi/documentation/files/1888662/BCM2837-ARM-Peripherals.-.Revised.-.V2-1.pdf.

[13] Raspberry Pi. *Tutorial oficial de instalación de Raspbian en la tarjeta MicroSD*. https://www.raspberrypi.org/downloads/.

[14] Jesús Carretero Pérez. *Sistemas operativos : una visión aplicada*. McGraw-Hill / Interamericana de España, 2 edition, 2007.

[15] Raspbian. *Página principal de Raspbian*. https://www.raspbian.org/.

[16] Vincent Sanders. *Sección de un artículo sobre arrancar un kernel ARM de Linux que describe los ATAGS*. http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html#appendix_tag_reference.

[17] Jake Sandler. *Building an Operating System for the Raspberry Pi*. https://jsandler18.github.io/.

[18] Andrew S. Tanenbaum. *Modern Operative Sistems*. Prentice Hall, 4th edition, 2014.

Agradecimientos

Agradecemos a nuestro director José L. Risco Martín por el tiempo cedido para orientarnos y ayudarnos a lo largo del desarrollo de este trabajo. También queremos agradecer a todos aquellos que nos han animado a seguir avanzando a raíz de la fuerte situación generada por la pandemia de COVID-19.